EXHIBIT B

Eurographics Symposium on Rendering (2007)
Jan Kautz and Sumanta Pattanaik (Editors)

Efficient Rendering of Human Skin

Eugene d'Eon, David Luebke, and Eric Enderton[†]

NVIDIA Corporation

Abstract

Existing offline techniques for modeling subsurface scattering effects in multi-layered translucent materials such as human skin achieve remarkable realism, but require seconds or minutes to generate an image. We demonstrate rendering of multi-layer skin that achieves similar visual quality but runs orders of magnitude faster. We show that sums of Gaussians provide an accurate approximation of translucent layer diffusion profiles, and use this observation to build a novel skin rendering algorithm based on texture space diffusion and translucent shadow maps. Our technique requires a parameterized model but does not otherwise rely on any precomputed information, and thus extends trivially to animated or deforming models. We achieve about 30 frames per second for realistic real-time rendering of deformable human skin under dynamic lighting.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

Accurate rendering of many real-world objects requires modeling subsurface scattering effects to capture translucent appearance. Examples of translucent materials range from milk and ketchup to jade, marble, and plastic. Many important materials consist of multiple translucent layers – notably many organic materials such as plant leaves and skin. Human skin in particular presents a challenging but crucially important rendering problem for photorealistic graphics. By approximating the physiological layers of skin – epidermis, dermis, and so on – as locally homogeneous thin slabs, researchers have achieved remarkably realistic renderings [DJ05,DJ06]. However, this realism comes at a cost: today's most accurate simulations of multilayer translucent materials typically require seconds or even minutes to render.

We present a novel and extremely efficient formulation of the multipole technique by Donner and Jensen [DJ05] for light transport through multi-layer translucent materials. Our technique enables real-time rendering of such materials and requires no precomputation. The key idea is to approximate diffusion profiles of thin homogeneous slabs as a linear combination of carefully chosen Gaussian basis functions. This representation greatly accelerates the computation of multi-layer profiles and enables improved algorithms for texture-space diffusion and global scattering via translucent shadow maps. We focus here on the specific application of these ideas to the rendering of human skin.

Most real-time methods capable of rendering translucent materials rely on precomputing light transport among points on the surface. This very general approach can capture a broad range of illumination effects, but requires fixing the geometry of the model at precomputation time (a notable exception: the zonal harmonics approach of Sloan et al. [SLS05] enables limited local deformations). Realtime rendering of translucent objects with no precomputation, necessary for scenes with fully general animation or deformation, has received much less attention. We improve and combine two such approaches: texture-space diffusion [BL03, Gre04], which provides an efficient estimate of local scattering, and translucent shadow maps [DS03], which approximates scattering through thin regions such as ears (Figure 6).

Specifically, we propose several extensions of texture-space diffusion to rapidly and hierarchically evaluate light diffusion within arbitrary deformable manifolds. The use of separable Gaussian kernels accelerates convolution of sur-

[†] {edeon, dluebke, eenderton}@nvidia.com

EXHIBIT C

Next Generation Skin Rendering

game|tech 2004

John Isidoro
Chris Oat
Jason Mitchell
ATI Research

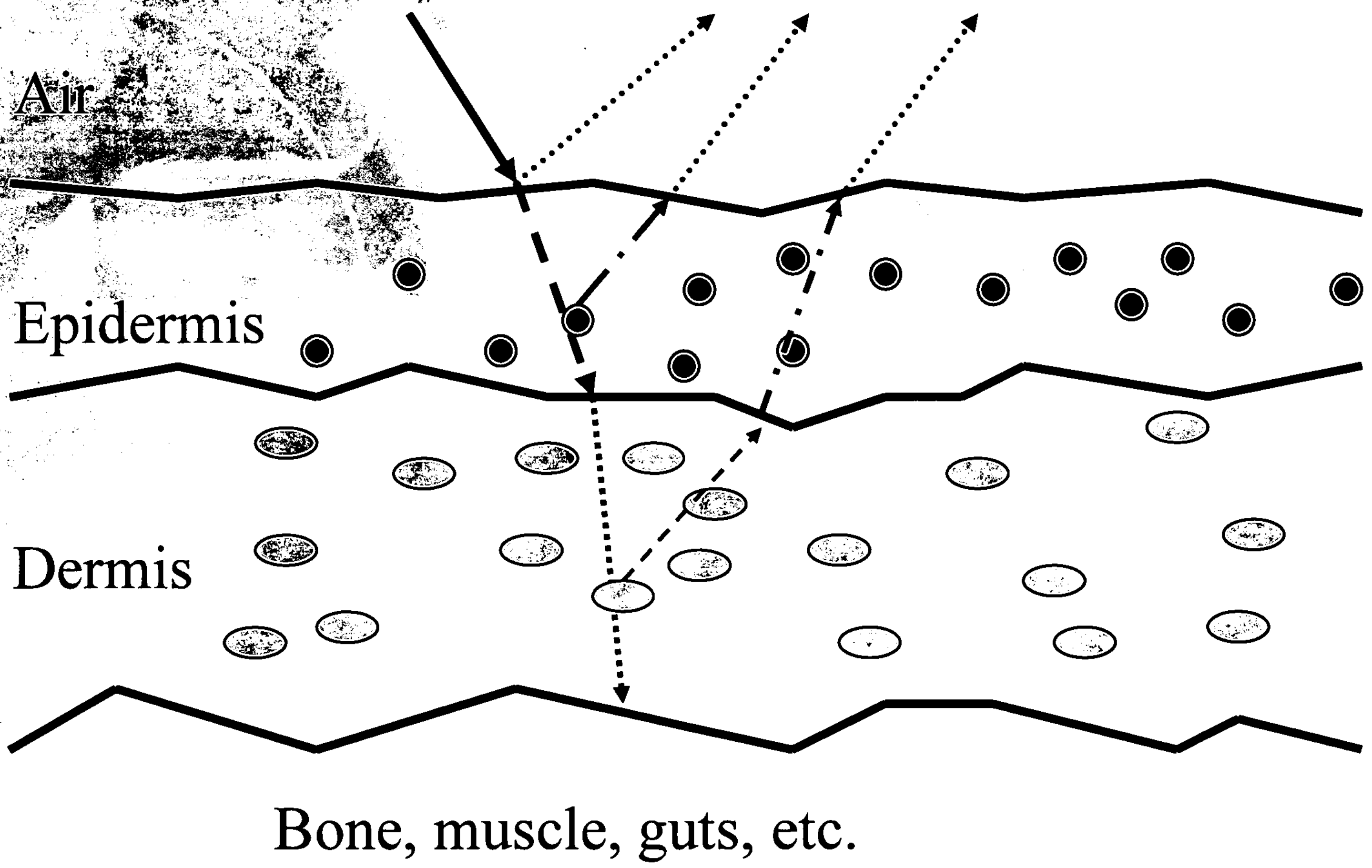
Overview

- Review
- Lighting Models
- Subsurface scattering
  - Texture Space Lighting
  - PRT
  - Irradiance Gradients
  - Zonal Harmonics
- Conclusion

Why Skin is Hard

- Most lighting from skin comes from sub-surface scattering
- Skin color mainly from epidermis
- Pink/red color mainly from blood in dermis
- Lambertian model designed for “hard” surfaces with little sub-surface scattering so it doesn’t work real well for skin

Rough Skin Cross Section



Basis for Our Approach

- SIGGRAPH 2003 sketch **Realistic Human Face Rendering for “The Matrix Reloaded”**
- Rendered a 2D light map
- Simulate subsurface diffusion in image domain (different for each color component)
- Used traditional ray tracing for areas where light can pass all the way through (e.g.. Ears)

Texture Space Subsurface Scattering

- From **Realistic Human Face Rendering for “The Matrix Reloaded” @** SIGGRAPH 2003
- From Sushi Engine

Current skin in Real Time

Real Time Texture Space Lighting

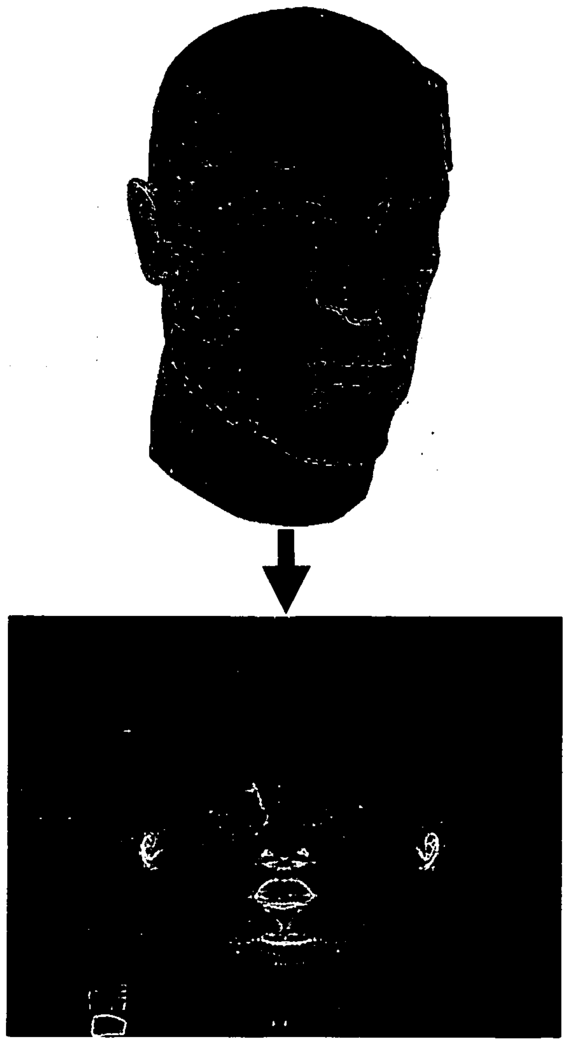
- Render diffuse lighting into an off-screen texture using texture coordinates as position
- Blur the off-screen diffuse lighting
- Read the texture back and add specular lighting in subsequent pass
- We only used bump map for the specular lighting pass

Standard lighting model

Blurred lighting model

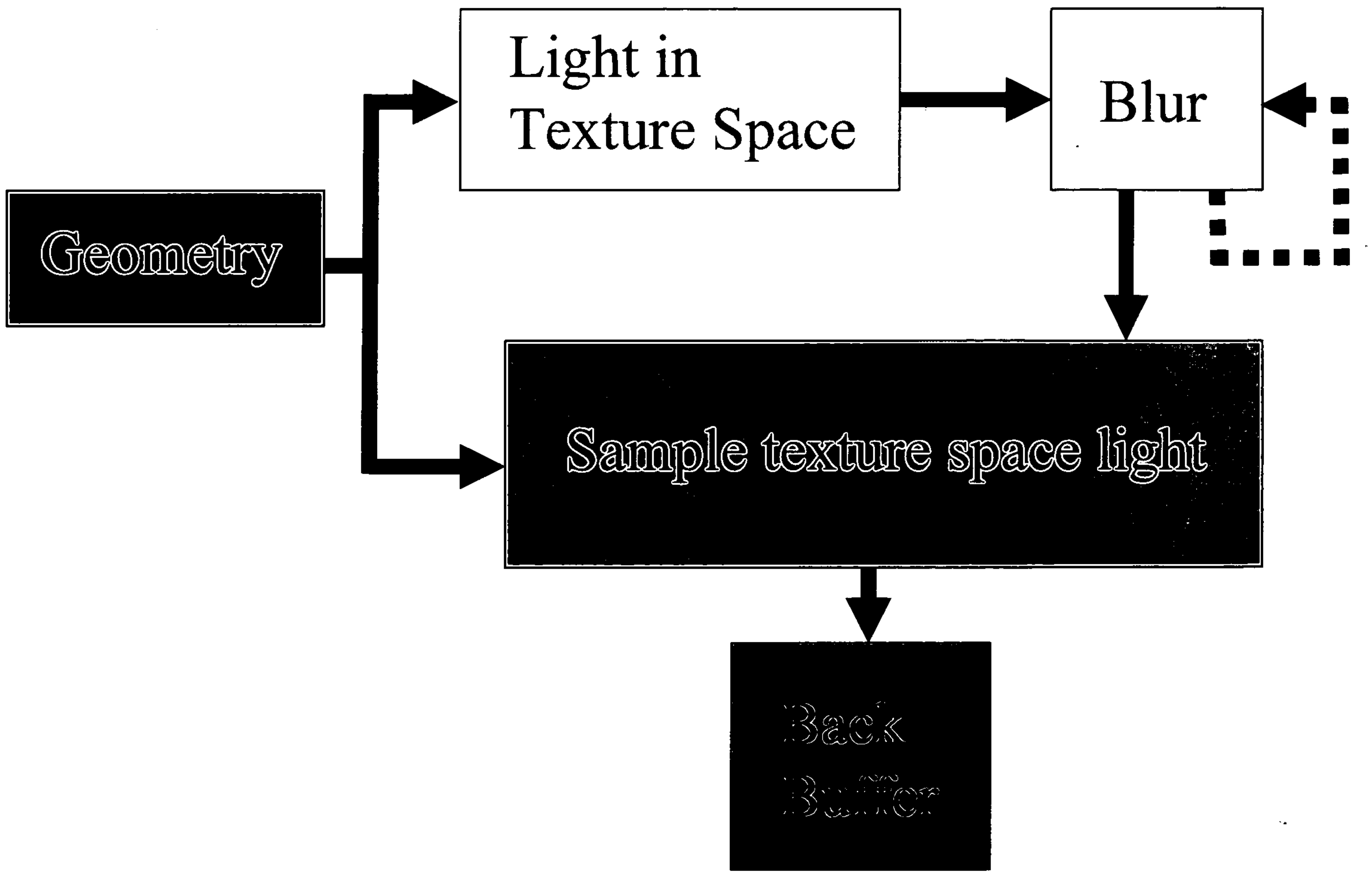
Texture Coordinates as Position

- Need to light as a 3D model but draw into texture
- By passing texture coordinates as “position” the rasterizer does the unwrap
- Compute light vectors based on 3D position and interpolate



game|

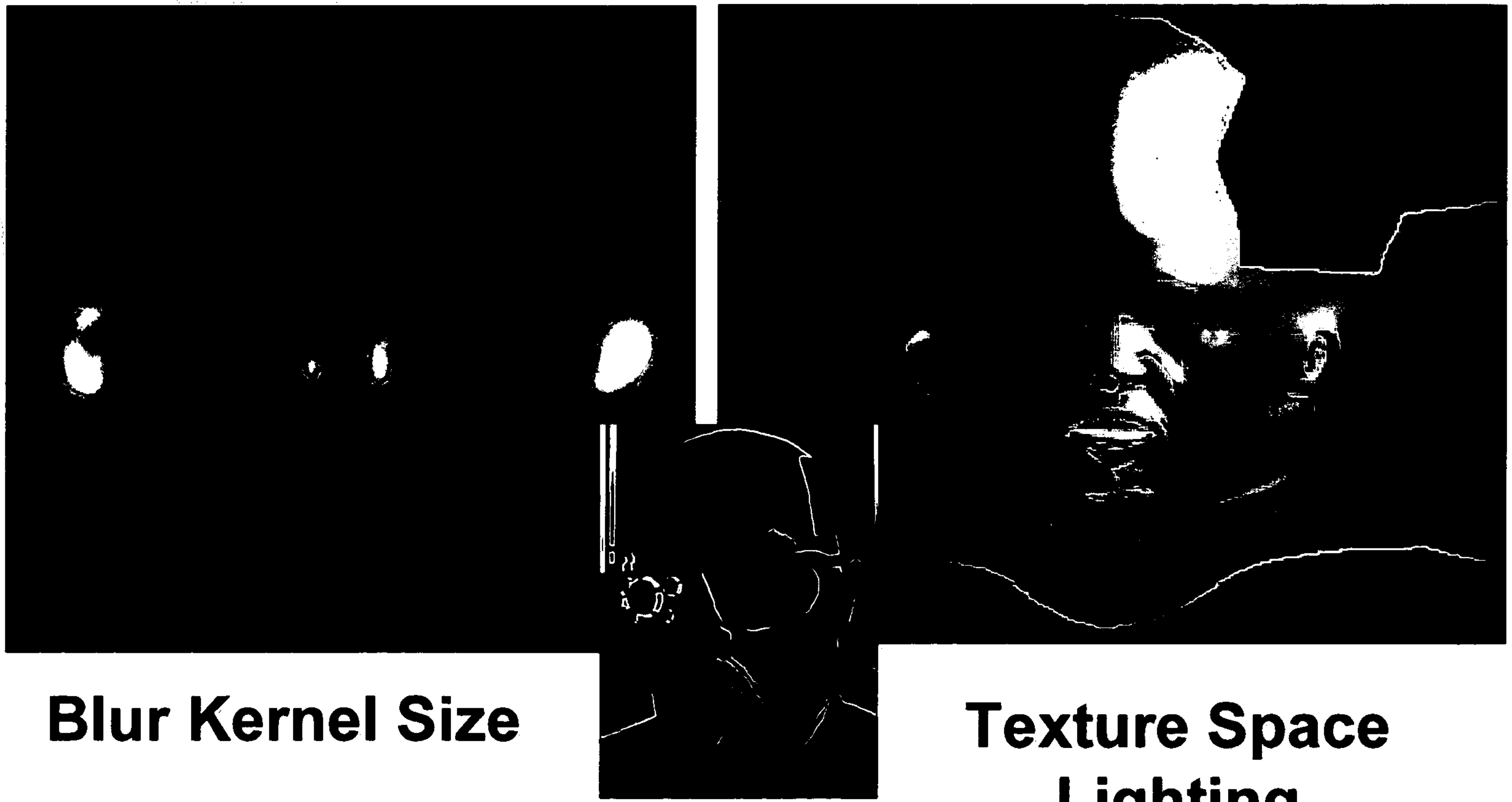
Basic Approach



Blur

- Used to simulate the subsurface component of skin lighting
- Used a grow-able Poisson disc filter (more details on this filter later)
- Read the kernel size from a texture
- Allows varying the subsurface effect
  - Higher for places like ears/nose
  - Lower for places like cheeks

Blur Size Map and Blurred Lit Texture


Blur Kernel Size

Result

Texture Space Lighting

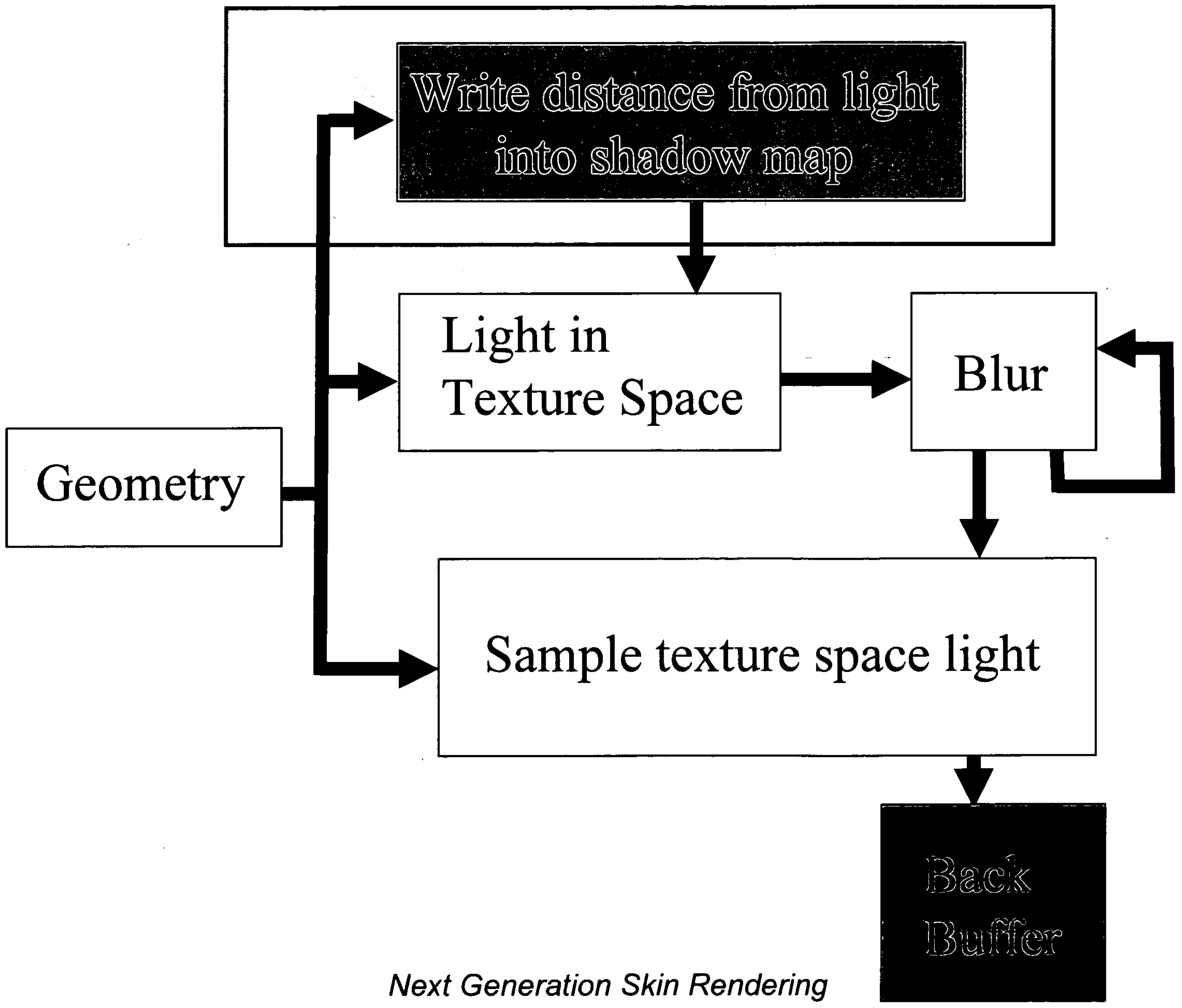
Shadows

- Use shadow maps
  - Apply shadows during texture lighting
  - Get “free” blur
    - Soft shadows
    - Simulates subsurface interaction
    - Lower precision/size requirements
    - Reduces artifacts
- Only doing shadows from one key light

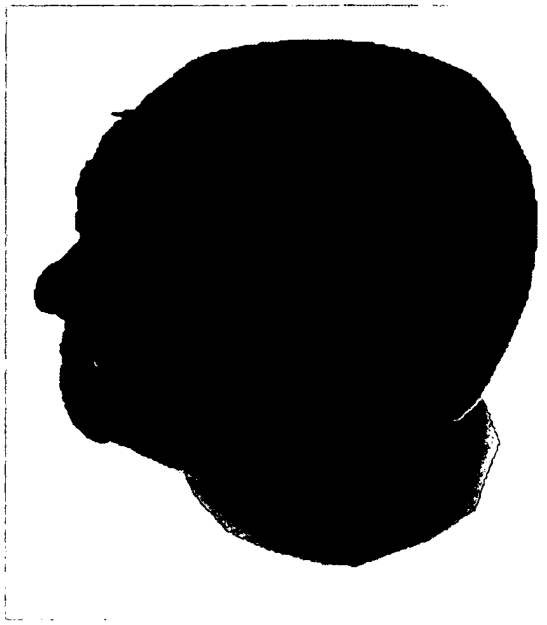
Shadow Maps

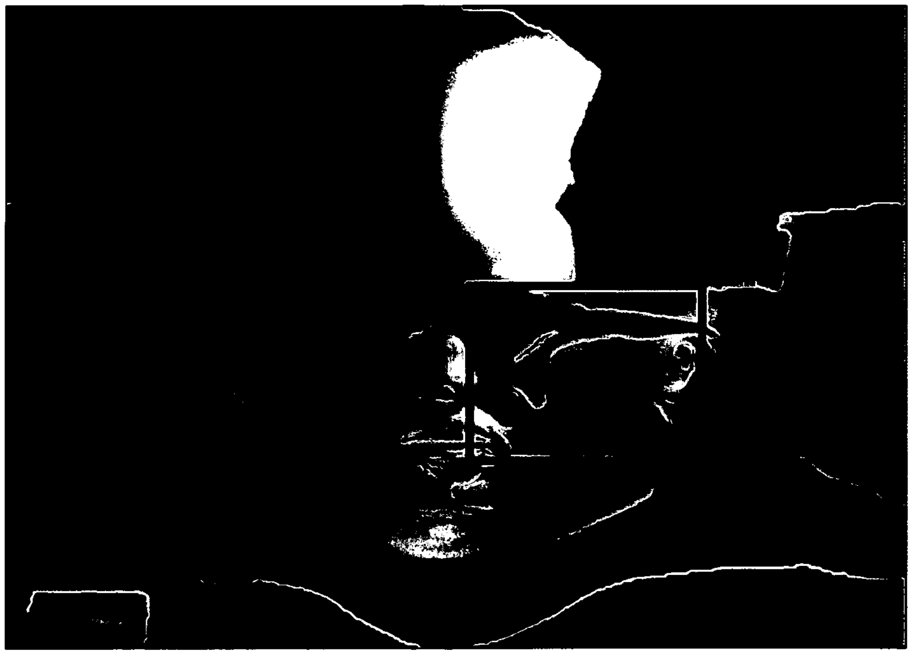
- Create projection matrix to generate map from the light's point of view
- Use bounding sphere of head to ensure the most texture space is used
- Write depth from light into off-screen texture
- Test depth values in pixel shader

Texture Lighting With Shadows


Write distance from light into shadow map
Light in Texture Space
Blur
Geometry
Sample texture space light
Back Buffer

Shadow Map and Shadowed Lit Texture


Shadow Map (depth)


Shadows in Texture Space

Result with Shadows

Using Early-Z for Culling

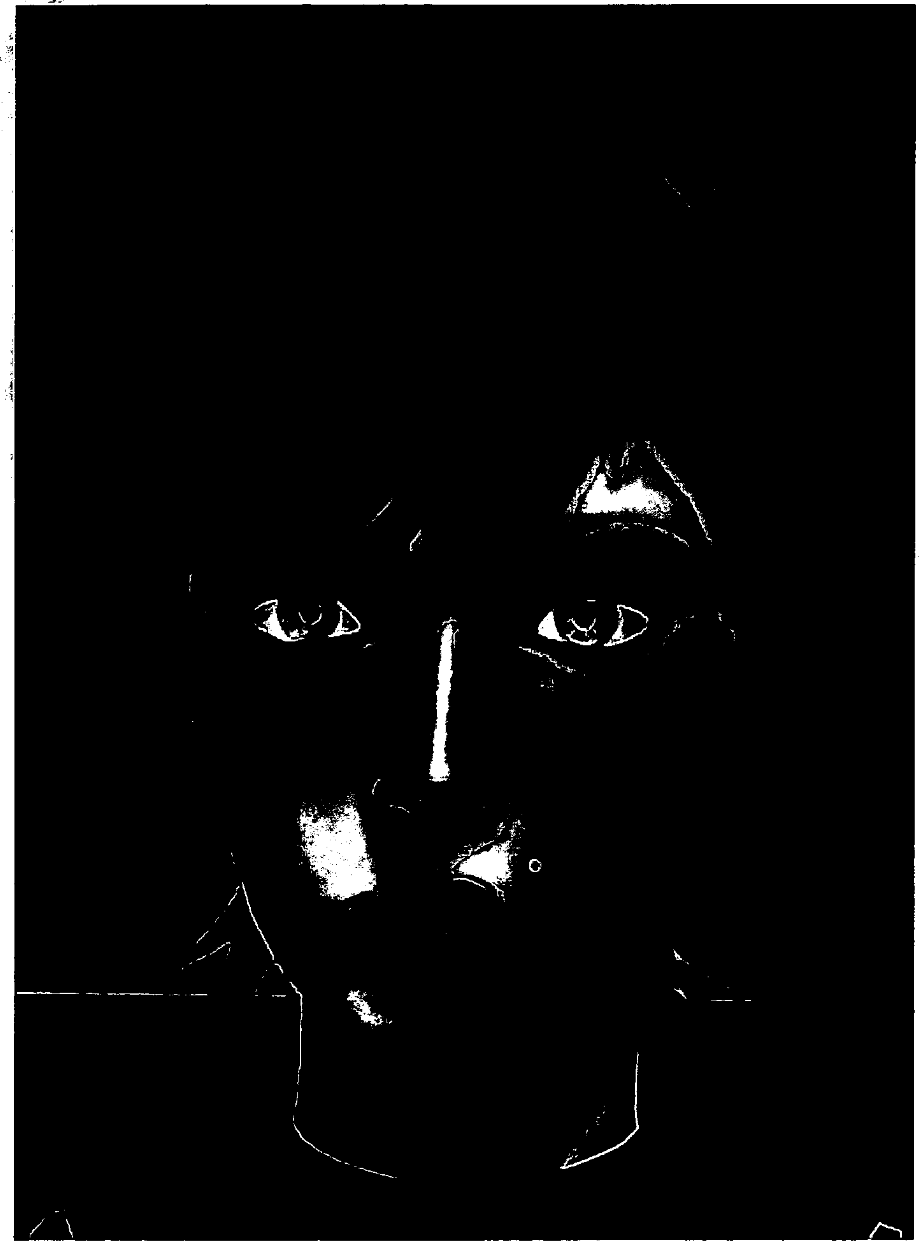
- Testing z-buffer prior to pixel shader execution
  - Can cull expensive pixel shaders
  - Only applicable when pixel shader does not output depth
- This texture-space operation doesn't need the z buffer for hidden surface removal
- Can store any value of Z buffer
- Use Early-Z to cull computations
  - Back face culling
  - Distance and frustum culling
- Set z buffer on lighting pass according to frustum, distance from viewer, and facing-ness of polygons
- Set the z test such that non-visible polygons fail Z test
- Reduces cost of image-space blurs in regions that don't need it

Back Face Culling

Back facing pixels
culled using early-z

ATI

game|



Next Generation Skin Rendering

Ruby2 Overview

- Overview of PRT lighting
  - Allows for sub-surface scattering, and global illumination effects.
- Irradiance volumes
  - Allows for changing incident lighting as Ruby moves through the tunnel.
- Irradiance gradients
  - Allow for variation in the incident radiance over the Ruby's extent in the scene
- Combining PRT lighting with standard rendering techniques in Ruby2
- Combining Ruby1 and Ruby2 style lighting
- Zonal Harmonics
  - Integrating this with skinning and morphing techniques.

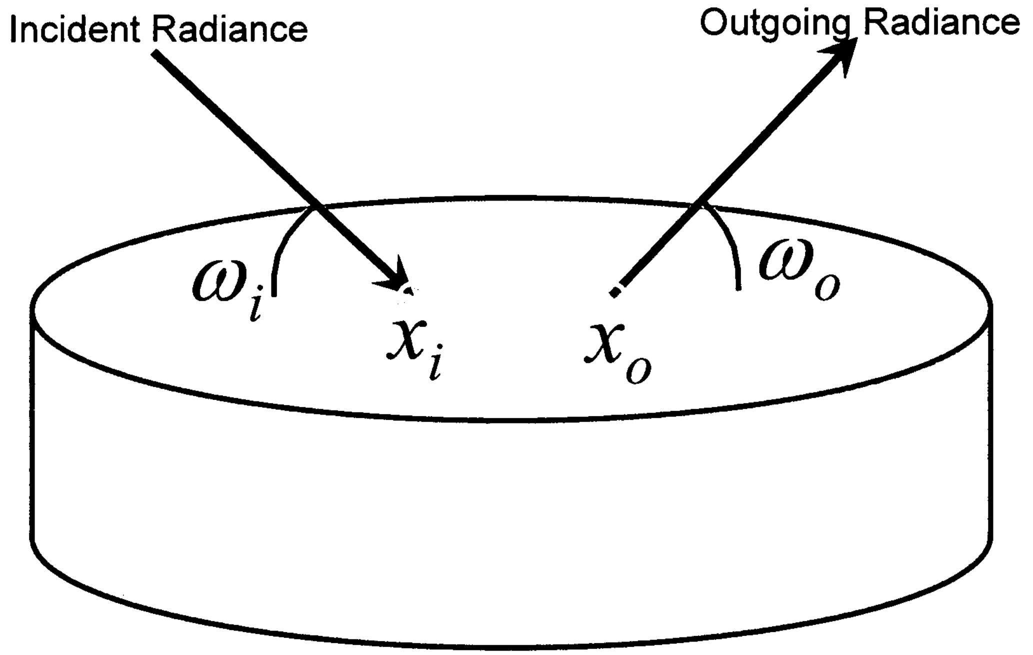
The Rendering Equation

$$L_o(x_o,\vec{\omega}_o)=\int_A\int_{2\pi} L_i(x_i,\vec{\omega}_i)S(x_i,\vec{\omega}_i,x_o,\vec{\omega}_o)(\vec{n}\cdot\vec{\omega}_i)\partial\vec{\omega}_i\partial x_i$$

Outgoing Light Intensity = Incident Light Intensity * BSSRDF: bidirectional subsurface scattering distribution function. * Hemisphere Cosine Term

- To compute the outgoing light intensity for a point on the surface and outgoing direction…
- We compute an integral over the incident light from all directions ω_i for each point on the surface x_i.
- Of course, storing the full 8-dimensional BSSRDF is very expensive, so we make a few simplifying assumptions….

About the BSSRDF $S(x_i, \vec{\omega}_i, x_o, \vec{\omega}_o)$



- Describes how light incident on the surface become reflected, refracted and scattered into outgoing light for all directions and points on the surface.
- Takes into account the effects of visibility, surface normals, indices of refraction, reflective properties, and light transport within the material.
- Allows for global illumination effects, and subsurface scattering.

Simplification of the BSSRDF for PRT

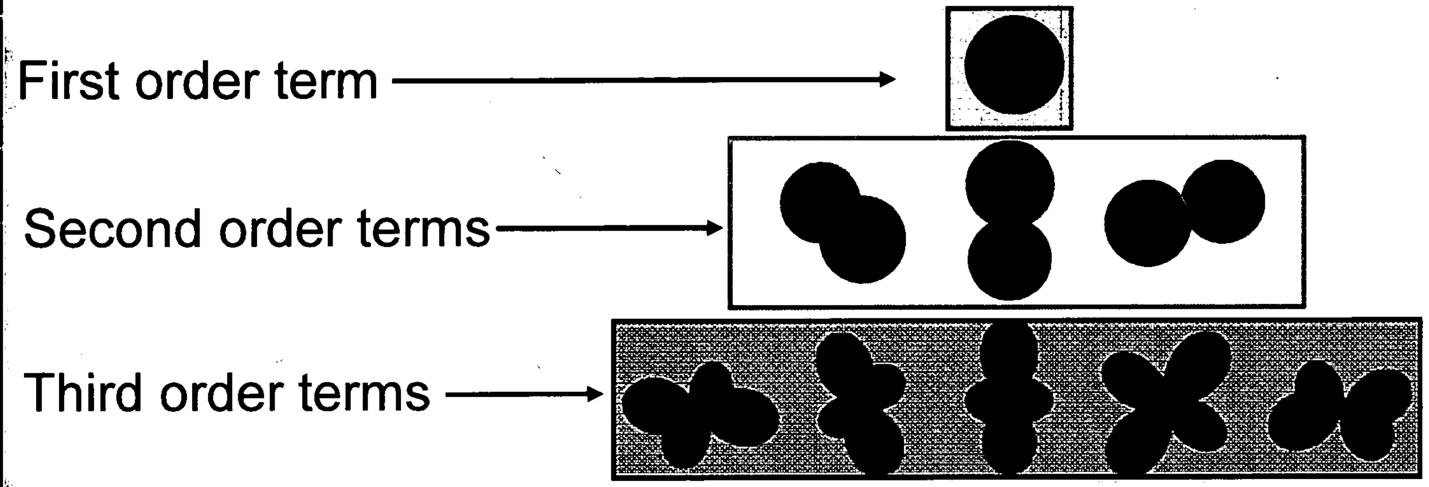
- Outgoing light is assumed to be diffuse, so no directional component is needed.

$$S(\cancel{x_i}, \vec{\omega}_i, x_o, \cancel{\vec{\omega}_o}) \Longrightarrow S_{PRTDiffuse}(\vec{\omega}_i, x_o)$$

- Light sources are assumed to be far from the object, so incident radiance is approximated as solely a function of direction.
- This results in a simplified rendering equation:
  - Note that the integral is only over the direction of the incident lighting

$$L_o(x_o) = \int_{2\pi} S_{PRTDiffuse}(\vec{\omega}_i, x_o)(\vec{\omega}_i \cdot n_i) L_i(\vec{\omega}_i) \partial \vec{\omega}_i$$

SH Basis



$$< C_0, \; C_1, \; C_2, \; C_3, \; C_4, \; C_5, \; C_6, \; C_7, \; C_8, \; C_9, \; \dots \; C_n >$$

- Allows you to represent functions on the spherical domain.
- Series is infinite
  - Choose a range that fits our storage and approximation needs
  - (6th order for skin / 4th order for other stuff)
  - Each function in the truncated series is assigned to an element in a vector.
- Each element stores its associated SH function's contribution to the overall signal (basis weight)
  - Building your original (arbitrary) spherical signal out of a fixed set of scaled, predefined spherical signals
  - The larger the "fixed set" the closer the approximation will be

PRT lighting

$$L_o(x_o) = \int_{2\pi} \boxed{S_{PRTDiffuse}(\vec{\omega}_i, x_o)(\vec{\omega}_i \cdot n_i)} L_i(\vec{\omega}_i) \partial\vec{\omega}_i$$

$$L_o(x_o) = \int_{2\pi} \boxed{T(\vec{\omega}_i, x_o)} \boxed{L_i(\vec{\omega}_i)} \partial\vec{\omega}_i$$

$$L_o(x_o) = \boxed{T(s, x_o)} \cdot \boxed{L_i(s)}$$

- The simplified BSSRDF and hemisphere cosine terms are combined into a transfer vector, and represented in the SH basis per-vertex (or per-pixel).
- The incident light (think environment map) is also represented in the SH basis.
- Integrating over all incident light directions can now be computed as a series of dot products. (Shader friendly!)

Example Shader for PRT lighting

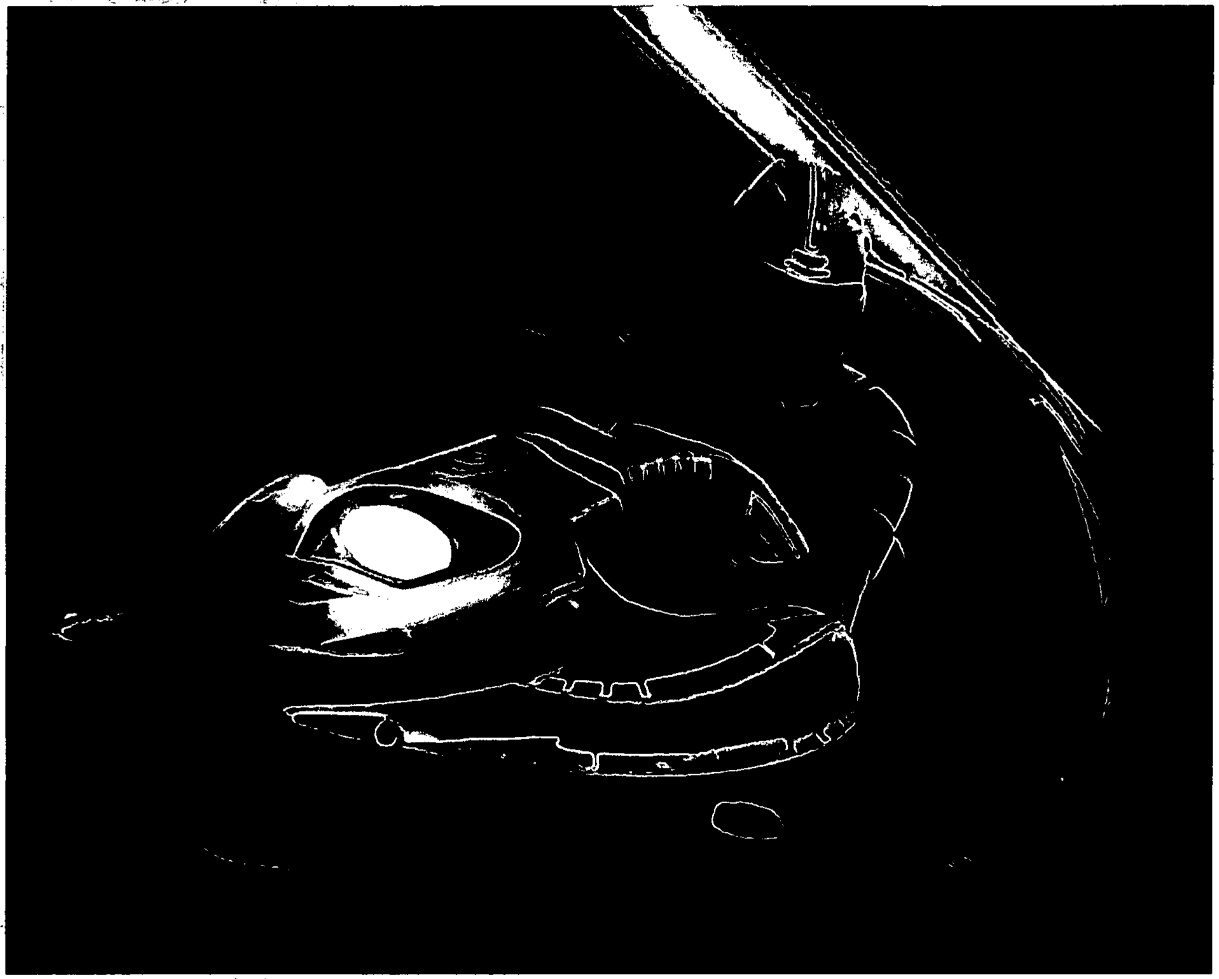
```
//----------------------------------------------------------------------
// HLSL code snippet for computing the PRT lighting integral via a sum of
//  dot products
//----------------------------------------------------------------------
for (int index = 0; index < (numSHCoeff/4); index++)
{
    o.cRadiance.r +=
      dot(i.vSHTransferCoef[ index] , g_vIrradianceSampleRedOS[ index]  );
    o.cRadiance.g +=
      dot(i.vSHTransferCoef[ index] , g_vIrradianceSampleGreenOS[ index] );
    o.cRadiance.b +=
      dot(i.vSHTransferCoef[ index] , g_vIrradianceSampleBlueOS[ index]  );
}
```

game|

Clustered PCA for PRT

- 6th order color PRT takes 108 coefficients
  - Too much data to store per-vertex or per-pixel
  - However, for most materials the PRT functions only span a small subset of the 108 dimensional vector space in a non-negligible way.
- Perform clustered PCA on the PRT data.
  - Derive a collection of representative transfer SH coeff. vectors that span the dominant portions of the subspace.
    - Generally between 4-24 PCA vectors (per cluster).
  - Store per-vertex (or per-pixel) weights to represent its own transfer vector as a weighted average of these vectors.
    - 4-24 weights per vertex (plus a cluster index)
  - Light the PCA vectors on the CPU using the incident radiance and pass the resulting colors into the shader constants.
  - In the vertex shader, compute a weighted average.
- CPCA acts as a form of lossy compression for PRT, but generally results in little loss in visual quality.

Case study: Ruby2

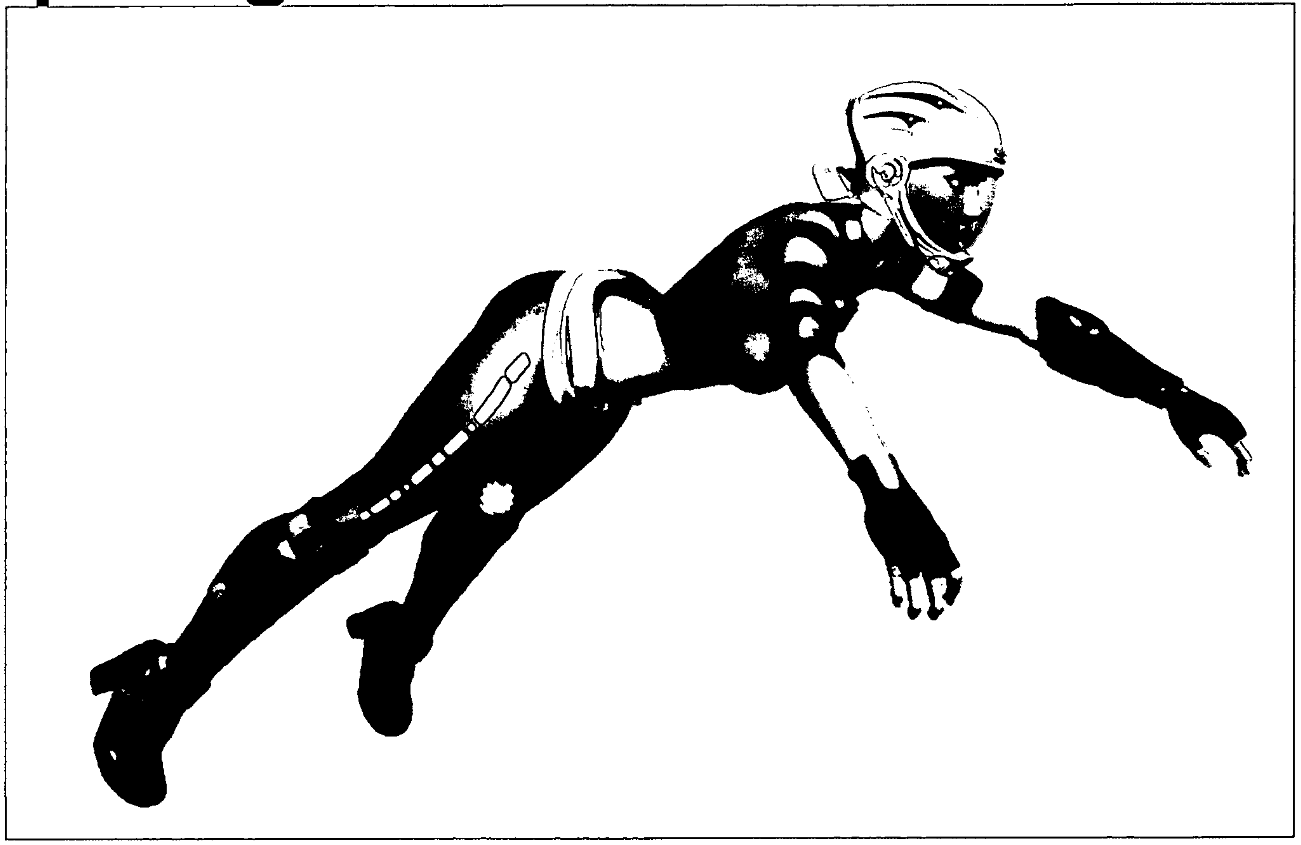


- Now we will show some methods to extend PRT techniques for motion though a complex lighting environment.

Ruby2 Improvements

- Canonical pose lighting
- Irradiance Volumes
  - Allows for changing irradiance throughout the scene.
- Irradiance Gradients
  - Allows for varying irradiance over a model.
- Integration with various material shaders.

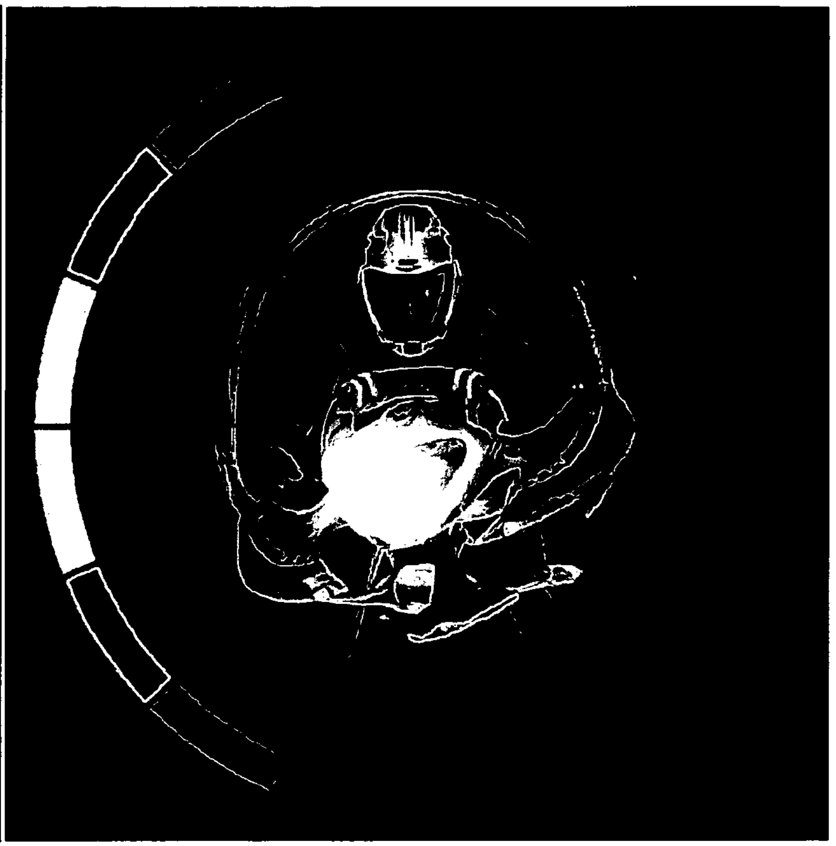
Computing PRT for a Canonical Pose



- In Ruby2 does not have a great deal of articulated motion, so pre-computing per-vertex PRT for the model for a single a canonical pose worked well enough.
  - The pose was chosen to minimize any shadowing effects that could change over the course of the demo.
  - PRT is mostly for ambient occlusion + sub-surface scattering effects
- Assumes no huge changes in occlusion/visibility.

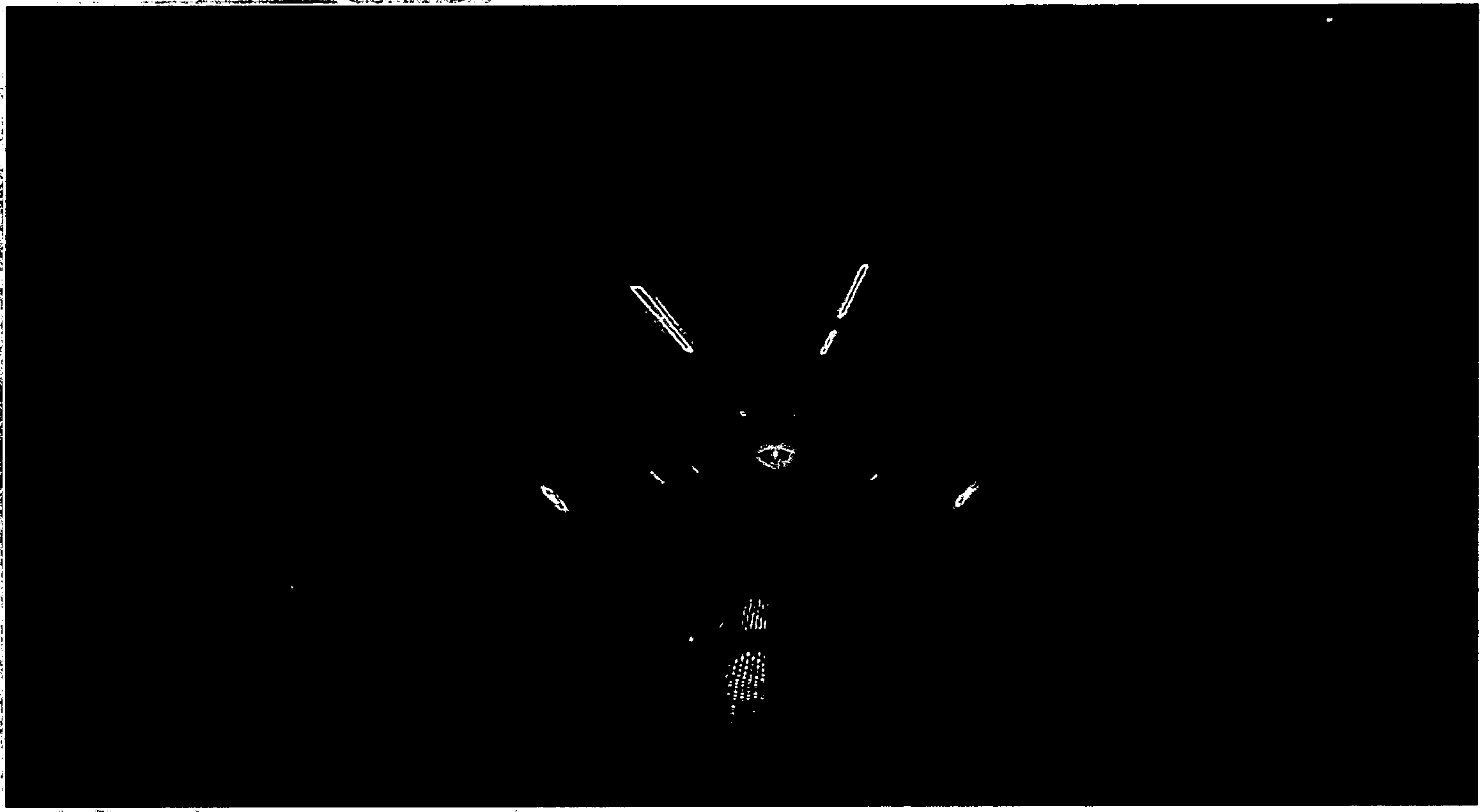
Incident lighting rotation

Lighting from above in world space, but Ruby is sideways on her bike.


Perform lighting in object space by rotating the world space incident lighting into canonical object space.

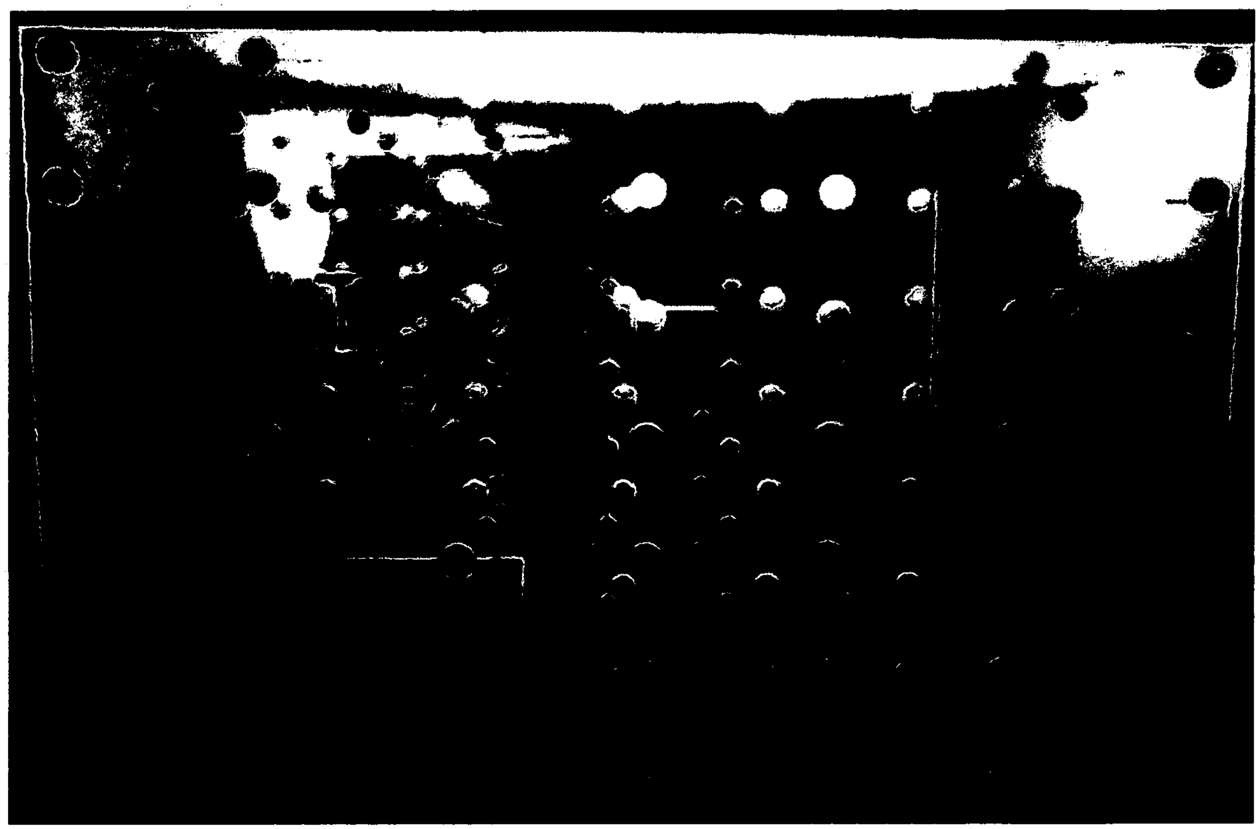
- Ruby + bike varies with respect to the tunnel, the WS incident lighting is rotated into object space of Ruby's canonical reference frame on the CPU.

Spatially varying illumination throughout the scene.



- A limitation of PRT based lighting in its basic form is that the light sources are assumed to be at infinity.
  - single lighting environment (irradiance sample) per scene.
- What can we do to get around this limitation?

Irradiance Volumes [Greger98]

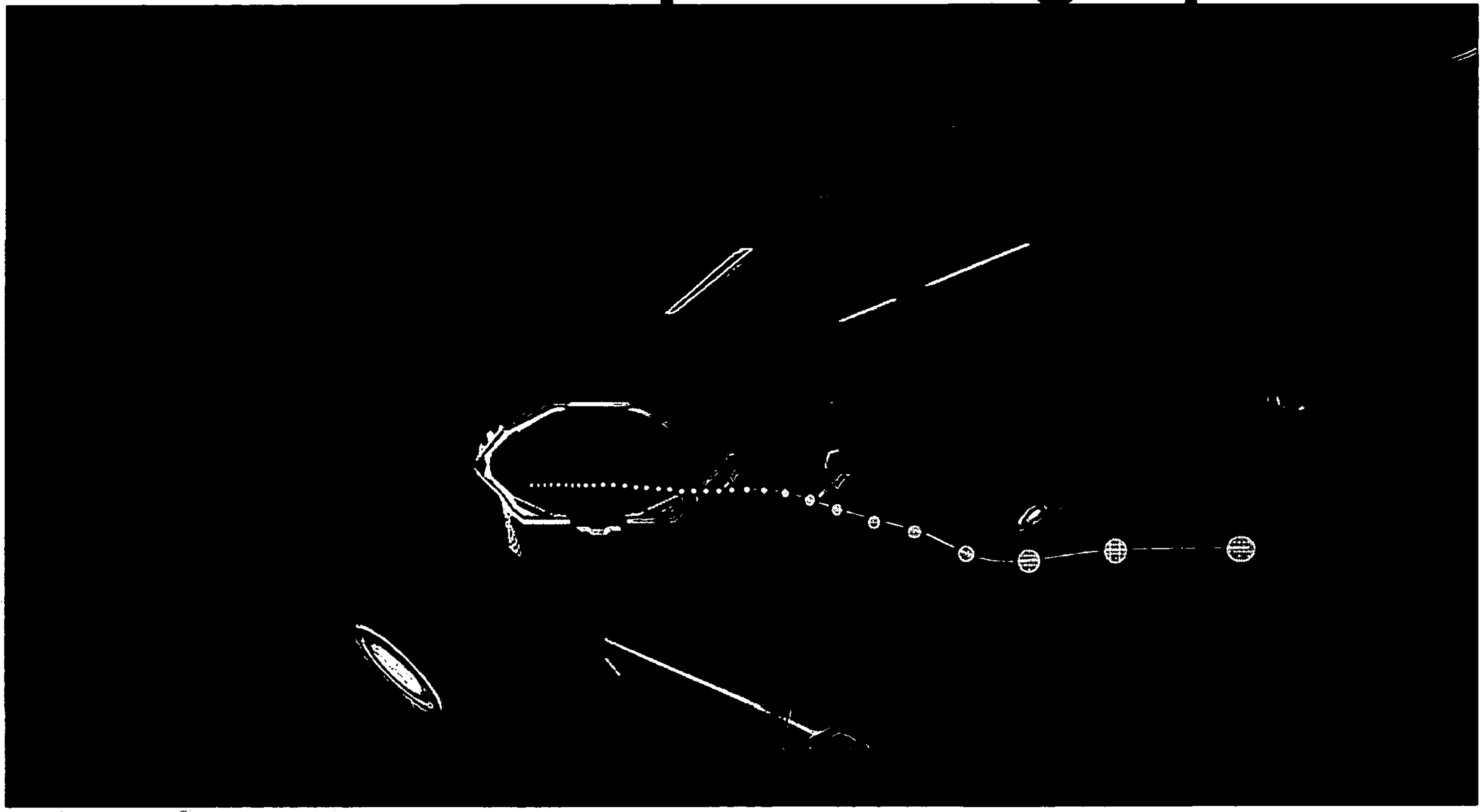


- A grid of irradiance samples taken throughout the scene
- For a point in the scene, the irradiance can be computed by tri-linear interpolation of the sampled irradiance within the scene.

Generating Irradiance volumes

- Sample irradiance by rendering lit scene and light emitters into a cube map at each point.
  - For SH based PRT lighting the SH coefficients are generated from the cube map.
  - Best performed at preprocess time
- Spacing between samples depends on detail in scene, and size of objects

Irradiance samples along a path

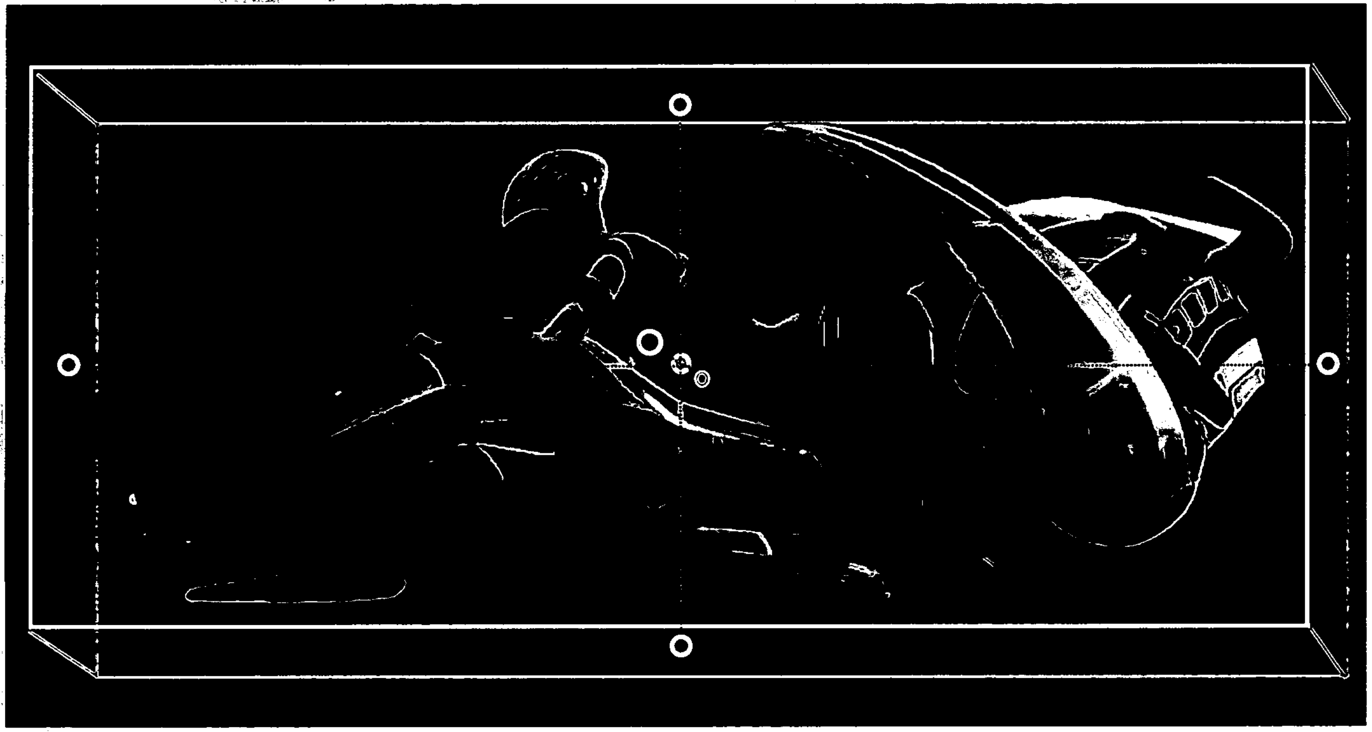


- In Ruby2, since the motion is constrained to the inside of a tunnel, thousands of irradiance samples are taken along the path her bike follows in the scene..
  - Also may be applicable to racing games..

Irradiance Gradients: Motivation

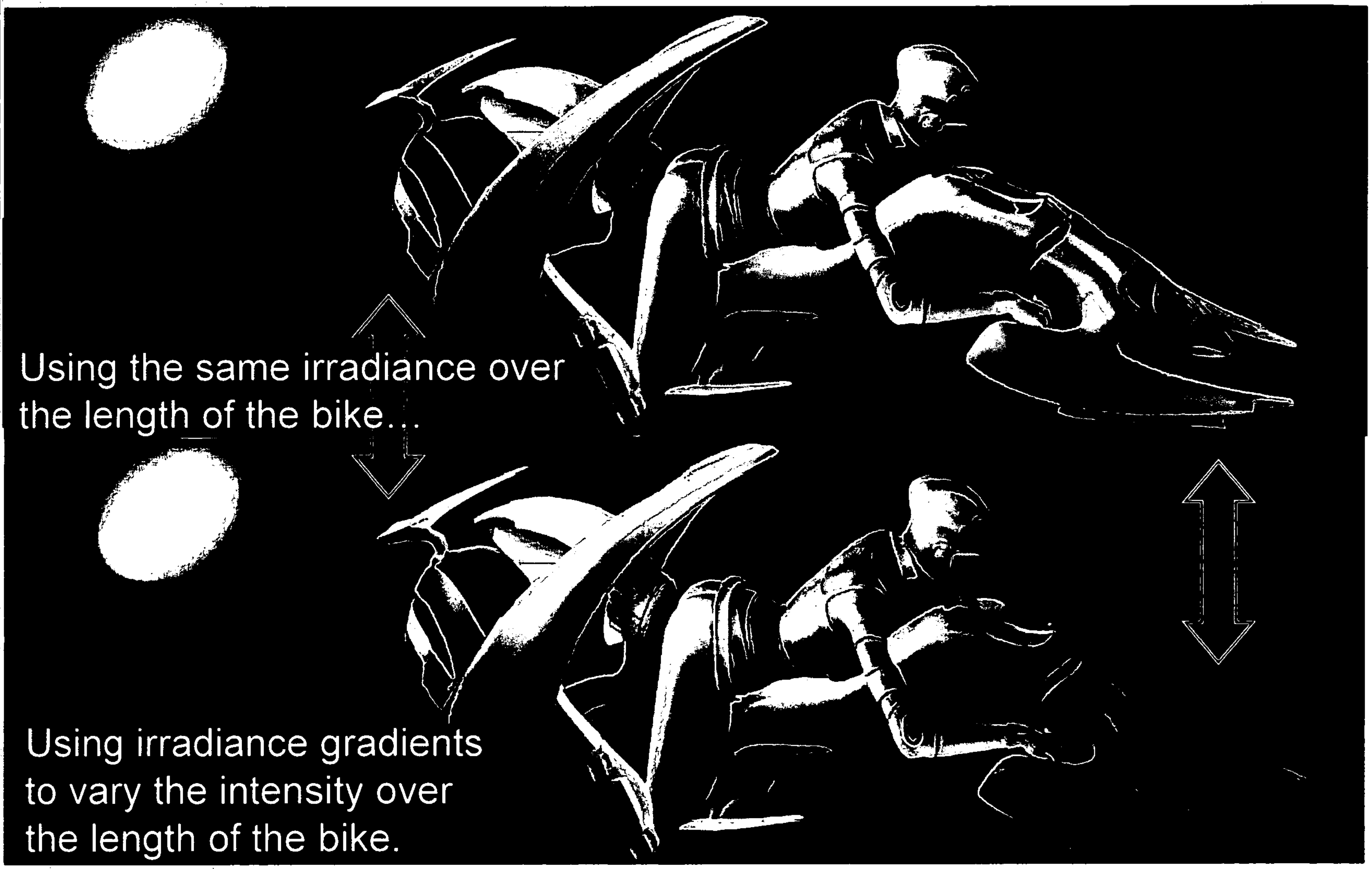
If irradiance varies greatly over an object due to nearby light sources, another possibility is to store irradiance gradients along with each irradiance sample. [Ward 92][Annen04]

Sampling for Irradiance Gradients



- For Ruby2 in preprocess we compute spatial derivatives in x, y, and z using finite differences.
  - Samples are placed at the center of each face of the world space bounding box of the object.
- Irradiance is computed for 6 different offsets and derivatives are computed using these offsets.
- At runtime, irradiance and its gradients are rotated into object space for each object being rendered.

Irradiance Gradient Examples

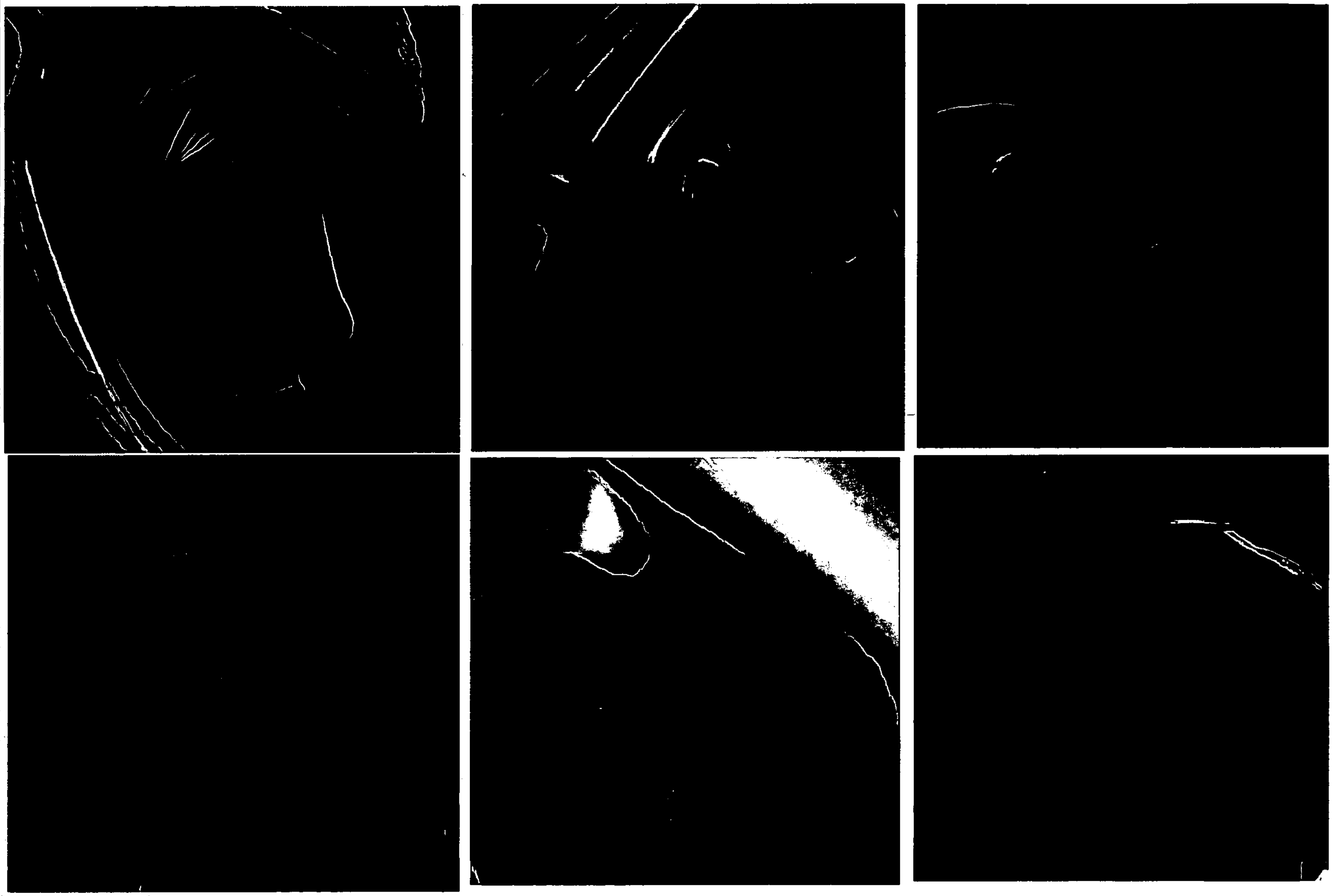

Using the same irradiance over the length of the bike…
Using irradiance gradients to vary the intensity over the length of the bike.

- Notice the variation in intensity over the length of the bike when using irradiance gradients.

Implementation Details

- Only the grayscale PRT coefficients are stored per vertex.
  - 6th order: 36 coefficients (9 float4 vectors)
- Color irradiance and irradiance gradients are stored in the vertex constant store.
  - (6th order: 36 coefficients * 3 channels * (1+3 gradients))
  - 27 vs constants (float4) for irradiance.
  - +81 vs constants for irradiance gradients.
- In the vertex shader..
  - First the point's irradiance is computed using the positional offset, center position's irradiance, and its gradients.
  - Then the PRT lighting integral is computed using dot products.

Using PRT lighting with other shaders

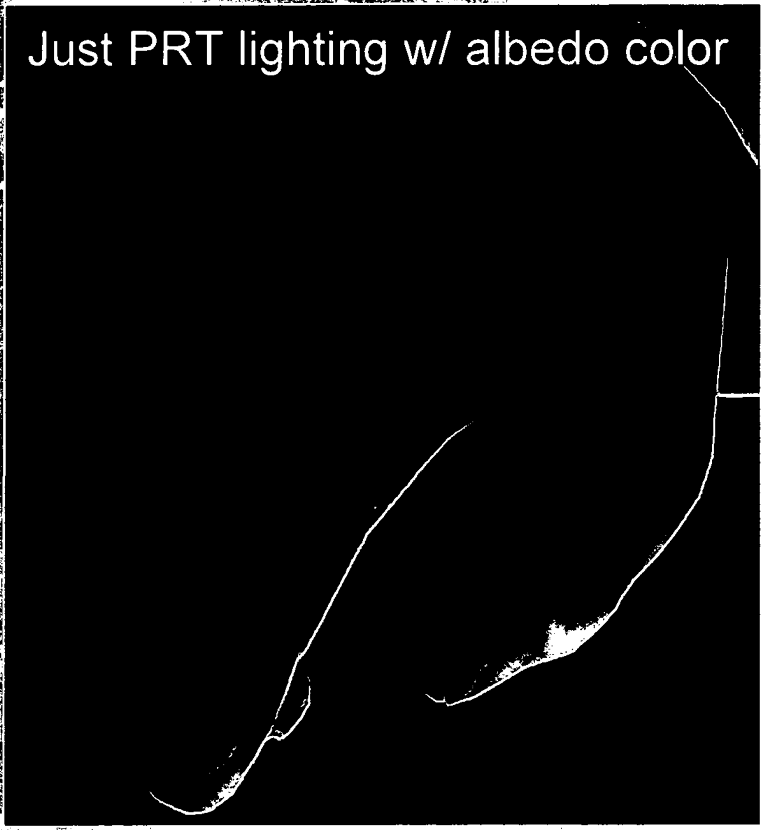


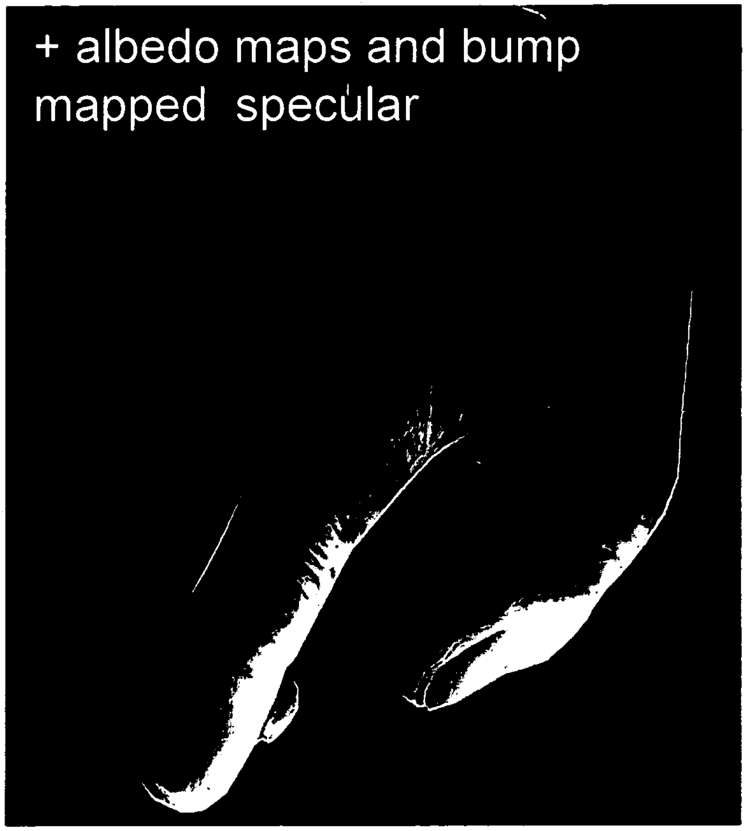
- PRT gives us diffuse and ambient lighting terms, but we would like to integrate these terms into more complex shaders…

Other information encoded in the SH basis for PRT

- 1st term in SH PRT basis acts as an ambient occlusion term
  - e.g. what percentage of the outside scene is visible from a particular point)
- Next 3 terms (2nd order) acts as a bent normal (aka shading normal)
  - e.g. what is the dominant direction of the visibility function for the point on the model)

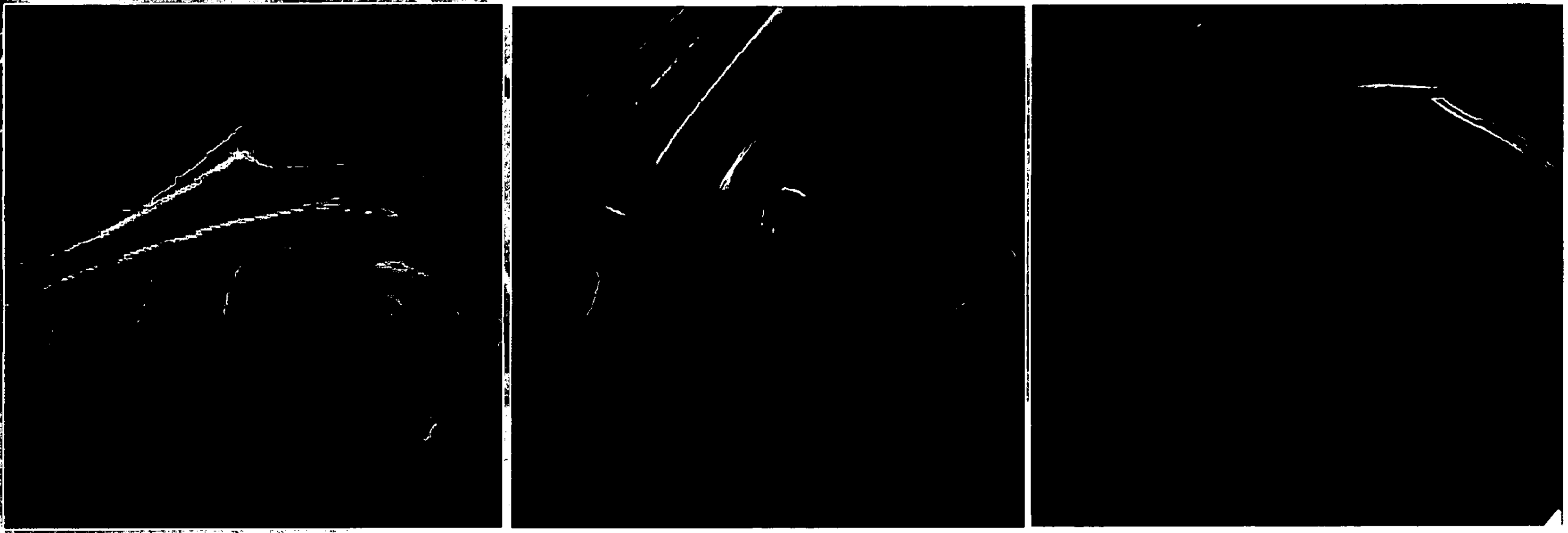
Example: Skin





- CPCA based
  - Uses color transfer vectors for reddening near thin regions.
- Modulate with albedo map
- Attenuate additive bump-mapped specular with ambient occlusion term.

Example: Bike paint shader



- Albedo map is modulated by PRT diffuse term
- Sparkle map: (high frequency bump map uses (N.V)^k)
- Specular lighting can be computed via dynamic cube maps and added.
- Specular is modulated with ambient occlusion (1st SH coeff. In PRT)
  - Reflections attenuated in occluded regions.

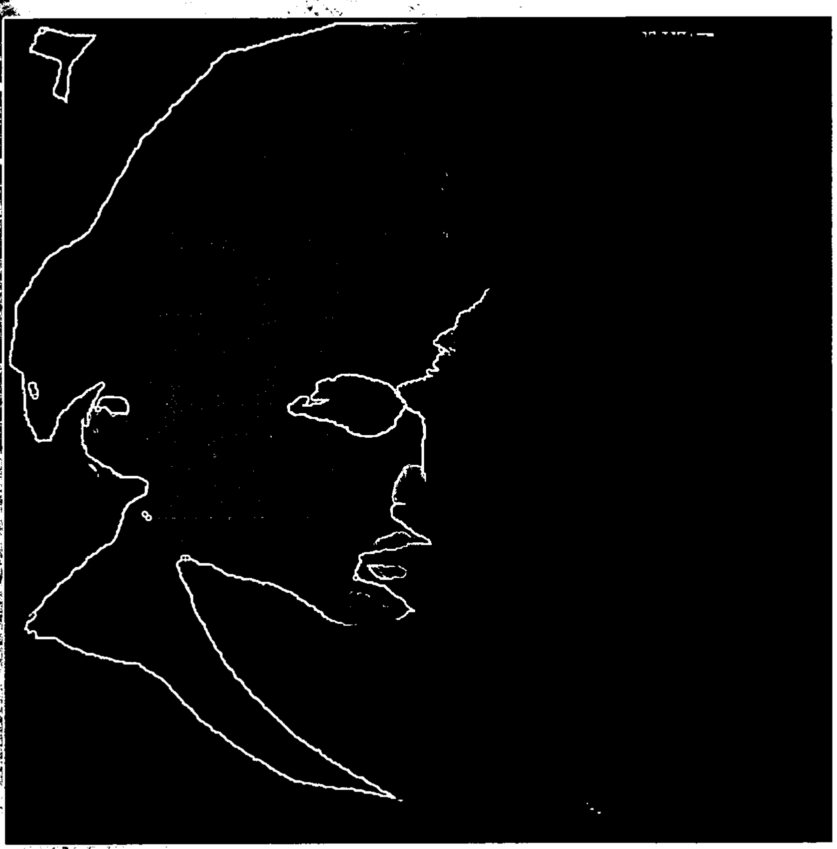
New Advances

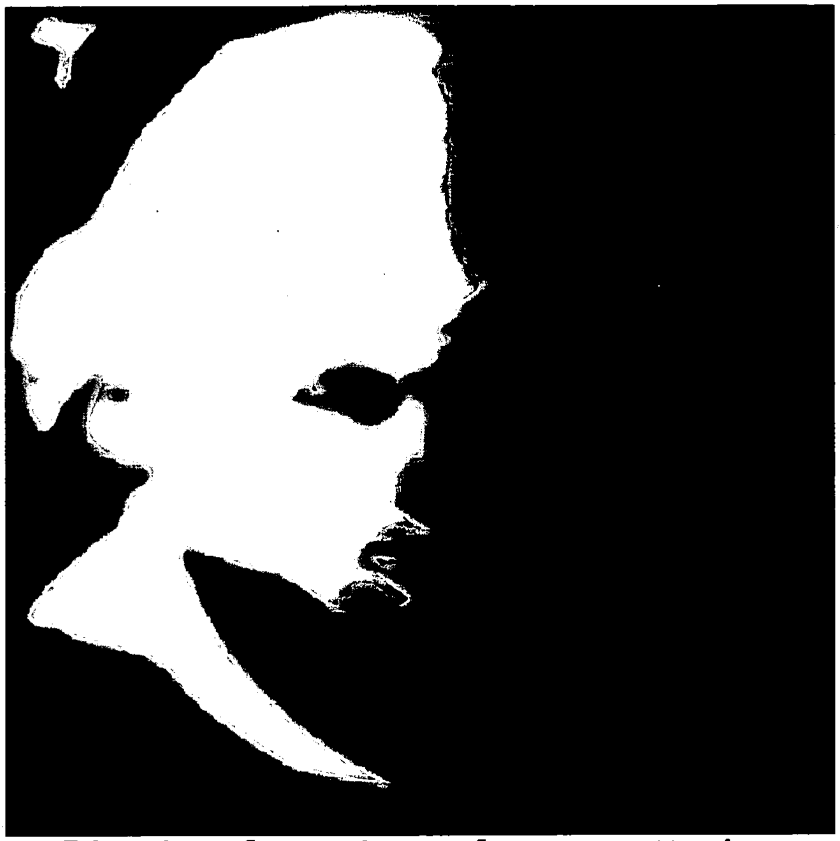
- Some results on combining PRT and standard lighting:
- Scattering in different wavelengths
- PRT for subsurface scattering term
- Zonal harmonics

How to Combine Techniques from Ruby1 & Ruby2

- Two Skin rendering approaches
  - Texture space lighting (Ruby 1)
    - Controls lighting from a single light source
    - High frequency variations in the lighting
    - Light source is generally nearby
    - Independent of material, and lighting model, and animation technique.
    - Shadow blur technique seens in previous section
  - Enhanced PRT based lighting (Ruby 2)
    - Can be used to get effects such as light shining through the ears, and nostrils.
    - Can model the effects of sub-surface scattering
    - Light sources assumed to be at infinity
    - Low frequency variations in lighting
    - Pre-process step
    - Animation unfriendly in basic form.

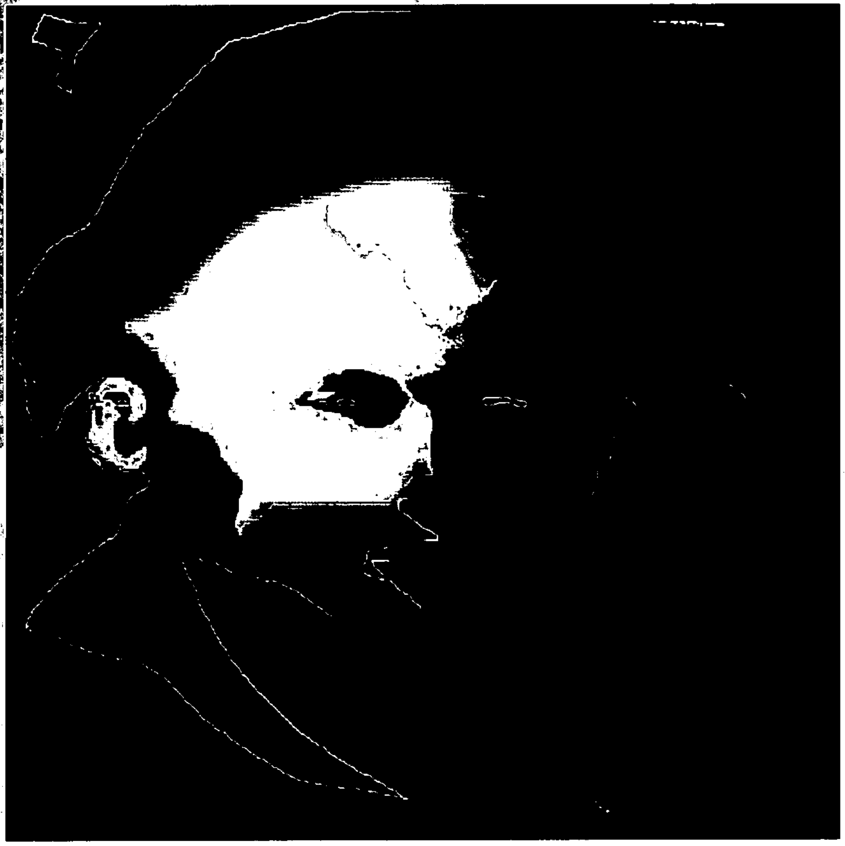
Blurring visibility

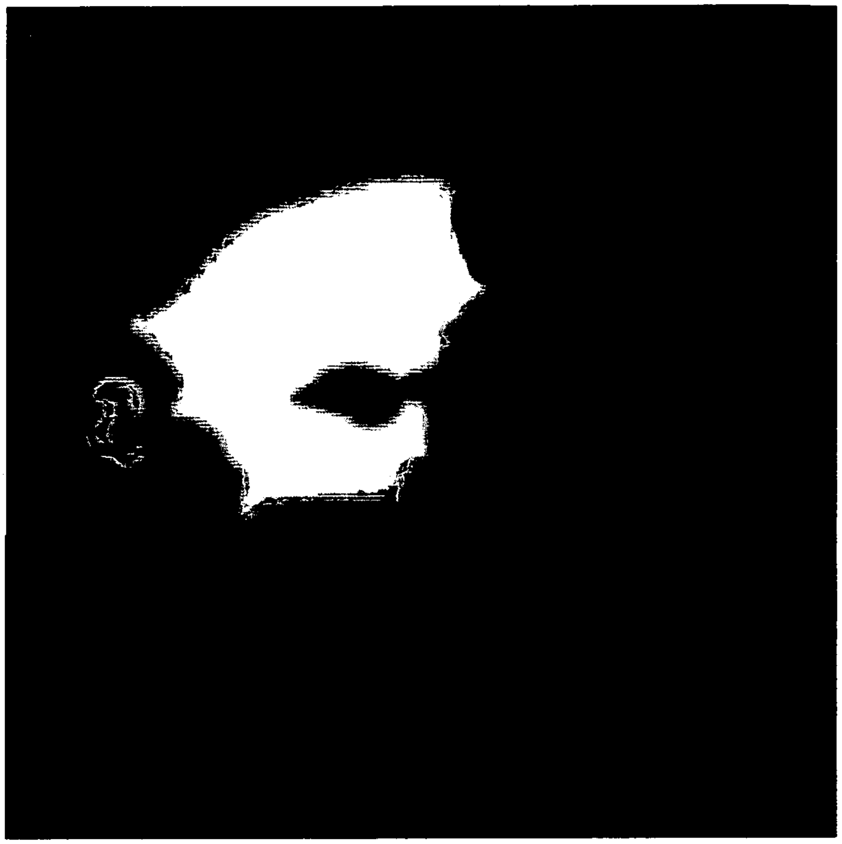

Visibility in light map space for 1 light


Blurring for sub-surface scattering effect

- Use shadow mapping to determine shadowed regions in light space
- Lightmap space blurring of visibility rather than lighting.
  - Each light uses one channel of a visibility map

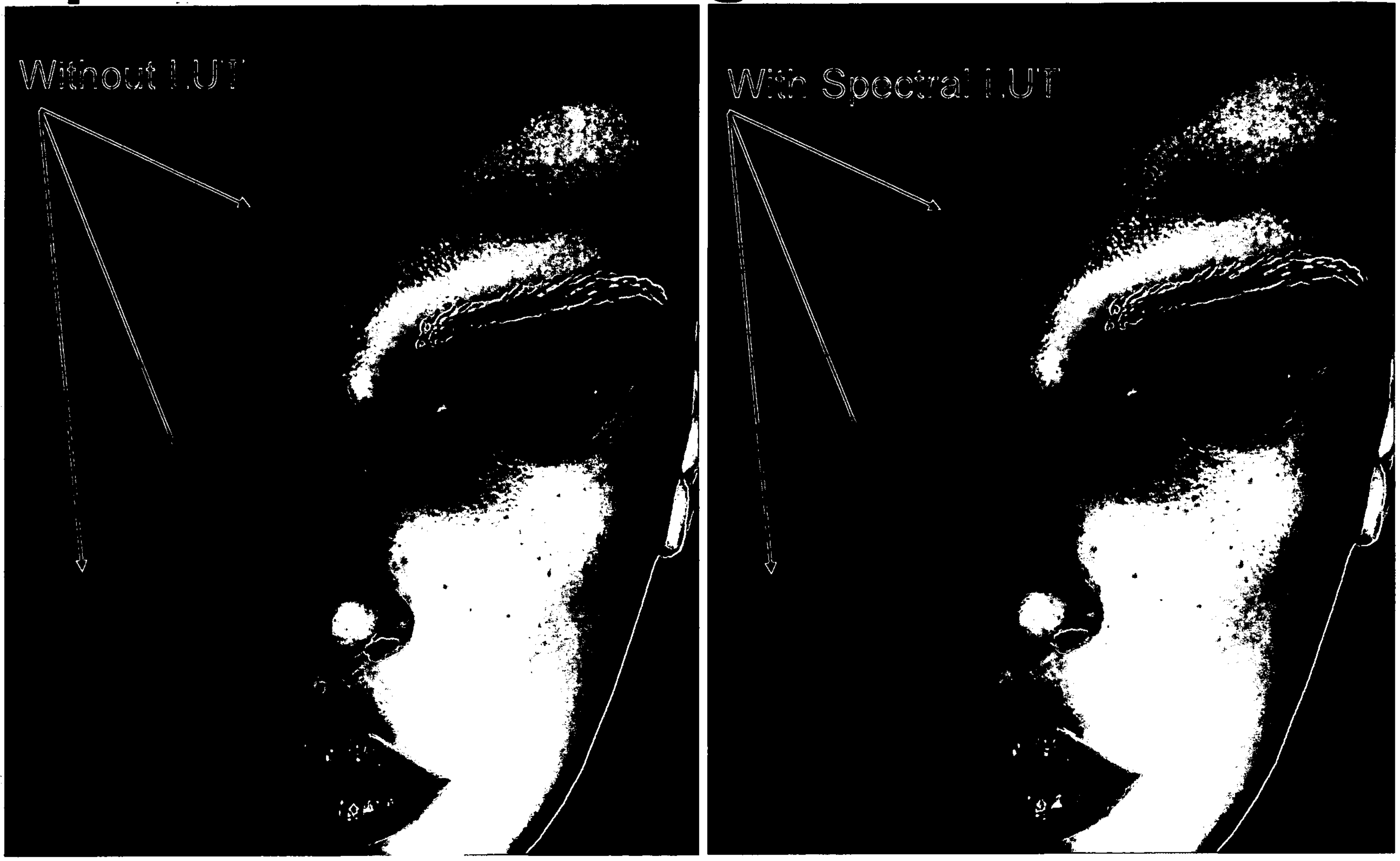
Blurring visibility


Visibility in light map space for 3 lights


Blurring for sub-surface scattering effect

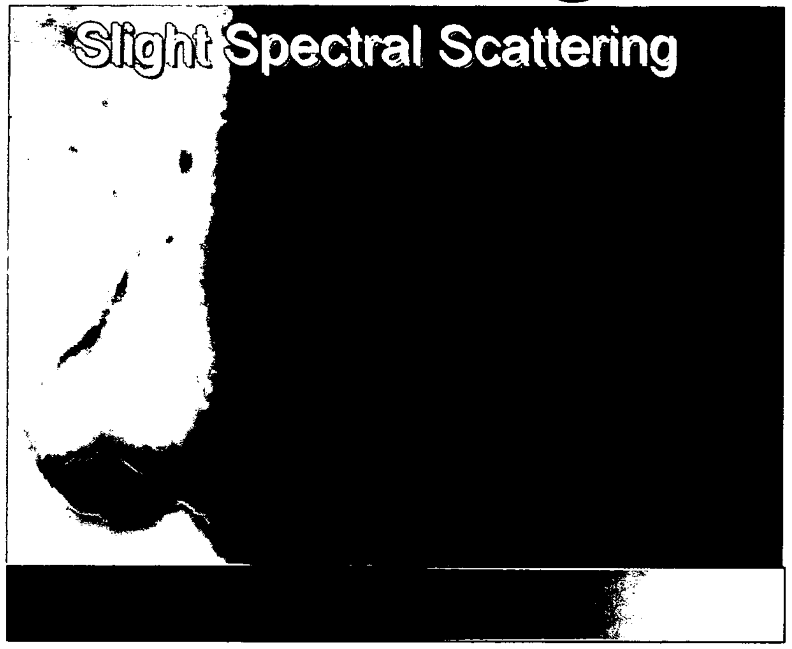
- Seen as each light uses one channel of a visibility map:
- We can blur shadows from four lights at a time if using an .rgba texture.

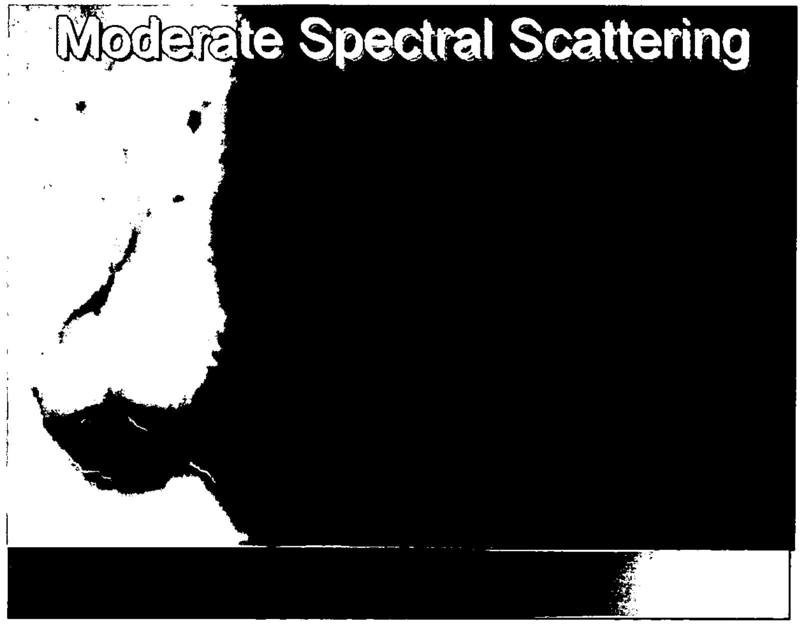
Spectral Scattering LUT

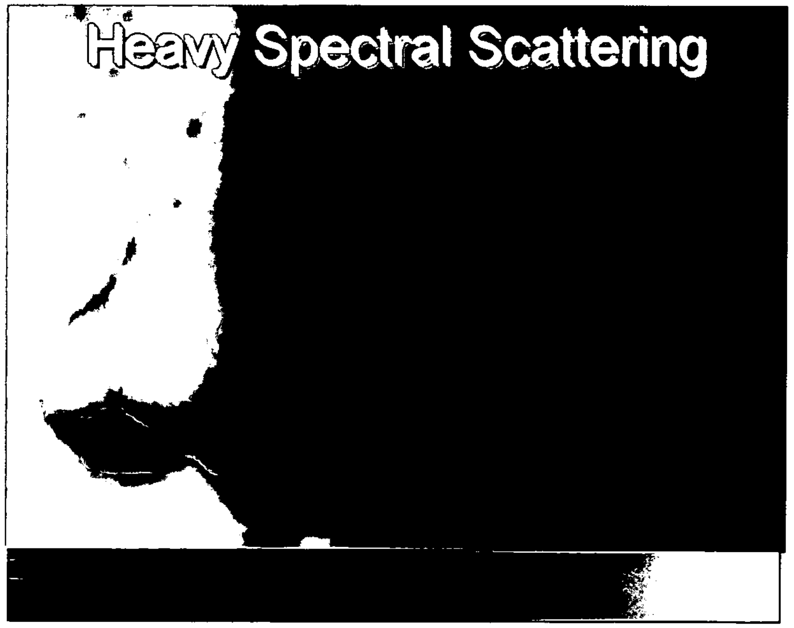


- Spectral scattering approach, 1D color LUT applied to blurred visibility edges to mimic effects of red light scattering in skin more than green light and blue light.

1D LUTs for Spectral Scattering

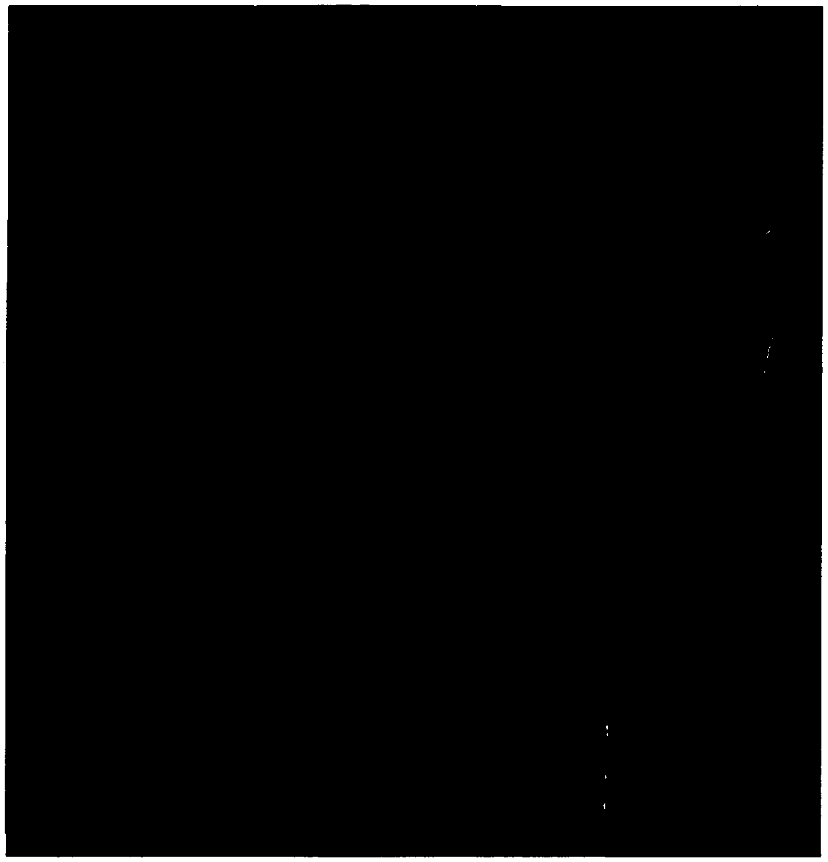






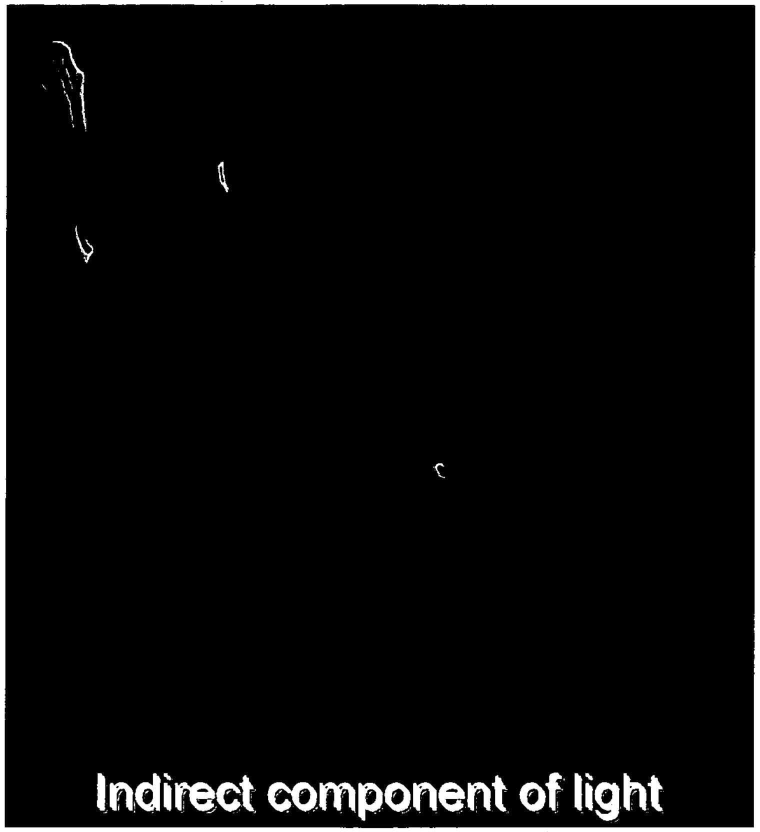
How to incorporate PRT lighting??

PRT only: Light from below


PRT only: Light from behind

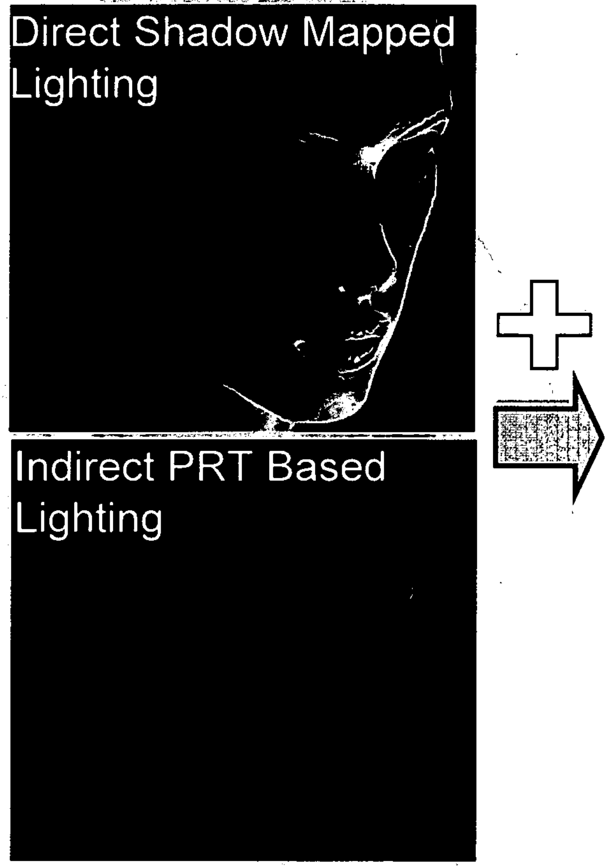
- We would like to apply the subsurface scattering effects using PRT to our shadow mapped lighting.
- Key idea: subtract direct illumination from PRT lighting, and add result to Ruby1 style shadow mapped lighting.

Indirect PRT lighting



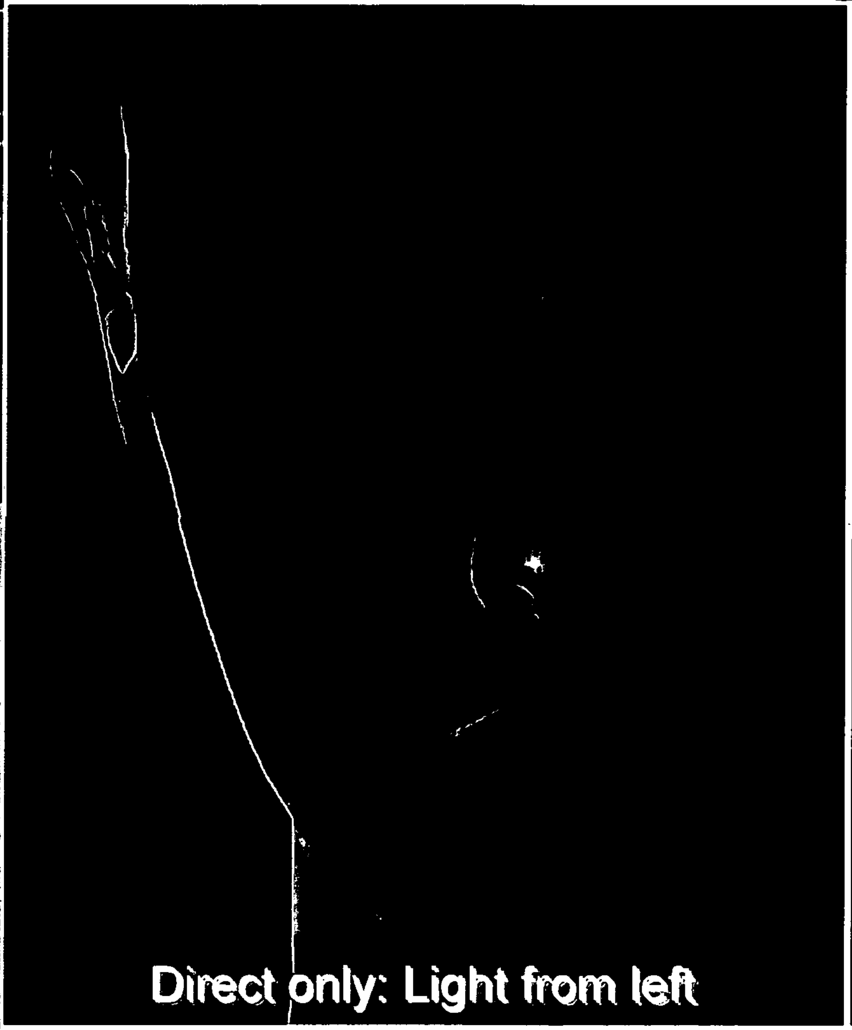
- Break incident light into per-light SH coefficients.
  - E.g. Multi-light PRT shaders
- Subtraction of $f(\mathrm{N} \cdot \mathrm{L})$ term from PRT lighting per light.
  - Attenuates light shining directly onto surface
- Use per-light rim lighting term $g(-\mathrm{V} \cdot \mathrm{L})$ to accentuate light bleeding through thin surfaces (backlighting).
  - In rim-lighting configuration, use PRT lighting as is for indirect lighting.

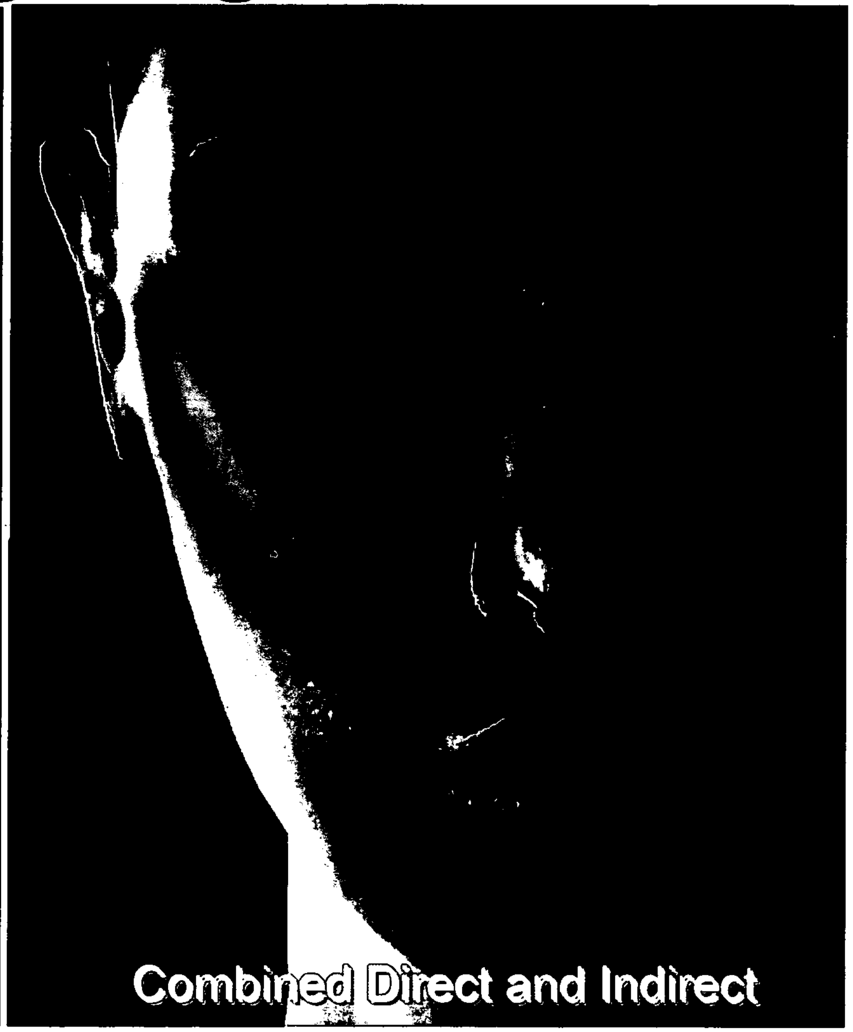
Direct and Indirect Illumination



- How can we combine the two:
  - Use shadow mapping with standard lighting to account for "direct" illumination
  - Use PRT based lighting to account for “indirect lighting” (subsurface scattered light)

Direct+Indirect Lighting Terms





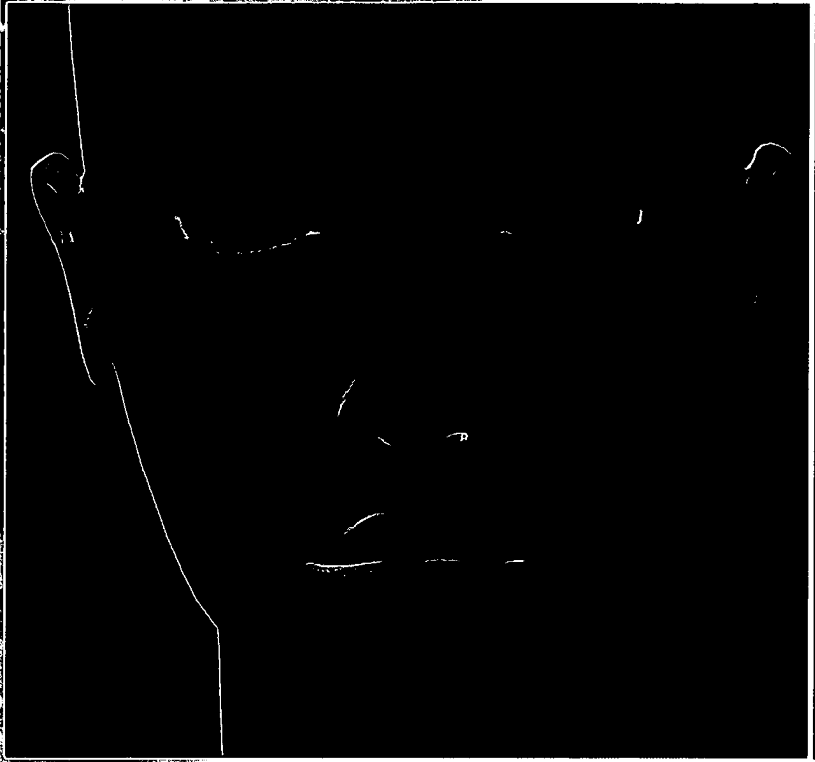
Allowing a greater deal of articulation..

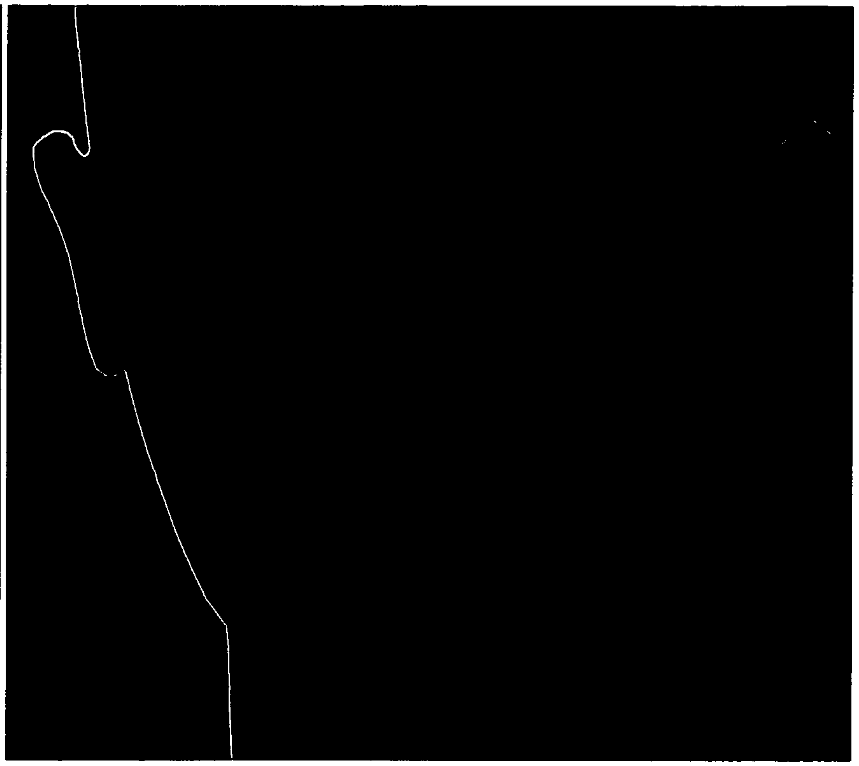
- If a greater degree of articulation is required..
  - One of the difficulties with using the SH coefficient based PRT, is rotating them in an efficient way.
    - Note that the basis functions for SH within each band are not just rotations of one another.
  - Possible to rotate lighting for each bone for skinning, but the results not easily fit in the VS or PS constant store.
  - For morphing one could imagine having different PRT for each morph target
    - But this would take additional vertex shader inputs, and limit the number of simultanously applied morph targets..
- Incident lighting and transfer vector need to be applied in the same coordinate system for SH PRT to be efficient…

Zonal Harmonics (ZH)

- Zonal harmonics: instead uses rotateable shading normal and per-band coefficients for transfer.
  - Coefficients control the shape of the BSSRDF lobe around that normal.
  - Same weight for all coefficients within a band.
  - Amplitude information is still there, but phase information encoded in the within band coefficients is replaced with a shading normal.
  - Per-vertex or Per-pixel
  - Can be computed from the SH transfer vector using D3D PRT tools function (CompConvCoefficients).
- Can rotate shading normal, this does not change the shape of the lobe about the normal.
- Approximation somewhere between diffuse SH irradiance lookup (equal per-band weights), and PRT using SH representation.

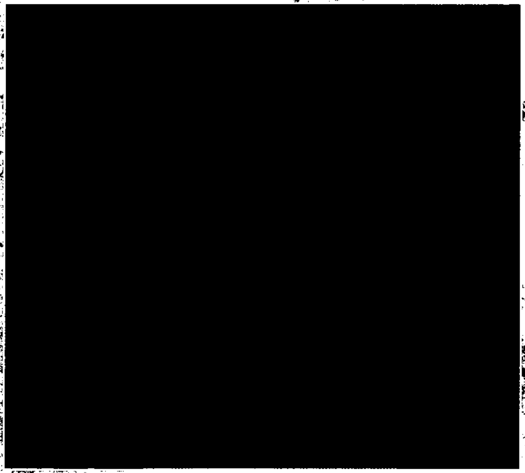
Shading Normals vs Surface Normals

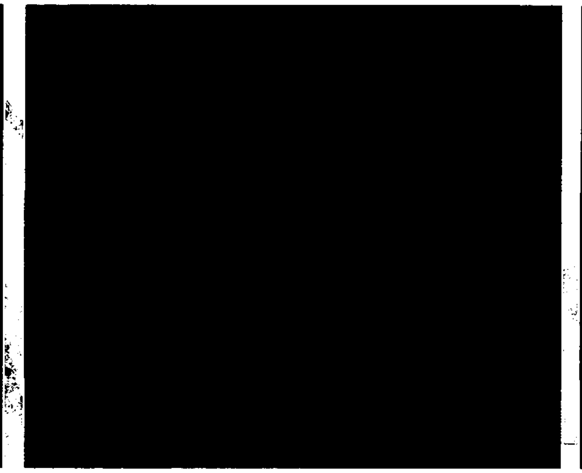

Surface Normals

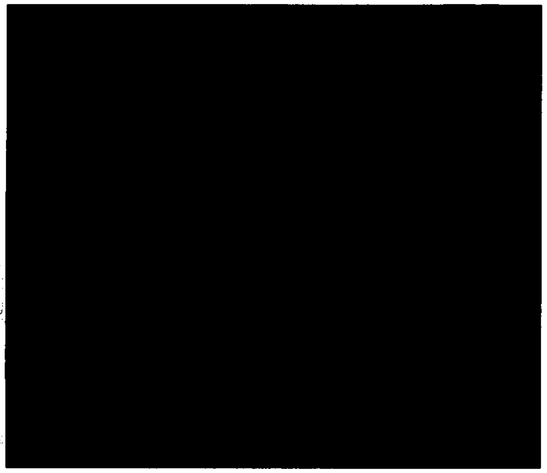

Shading Normals

- Notice how the shading normals vary less over the surface than the surface normals
  - Contributes to the overall “softness” of the appearance of the skin.
  - The variation of the shading normals over the surface is material dependant.

Zonal Harmonic Coefficients


ZH Coeff Band0


ZH Coeff Band1


ZH Coeff Band2

- Think of each band of the incident lighting as a band pass filtered version of the incident lighting.
- The zonal harmonics coefficents control the shape of the lobe about the shading normal used to sample the irradiance.
- For the skin material, the contribution of each band of the zonal harmonics decreases as frequency increases.
  - Analogous to a low pass filtering of the incident light.
  - Contributes to the smooth diffuse appearance of the material.
  - Intuitively, Band0 acts as an ambient occlusion term, and Band1 scales the contribution of the shading (bent normal).
  - For the skin material only the first 3 ZH bands were needed.

Example Shader for ZH PRT

```
//-----------------------------------------------------------------------------
// Constants for Linear + constant polynomials
// g_vCartSHConstB12 = {
//    1/(2*sqrt(pi)),   -sqrt(3)/(2*sqrt(pi)),   sqrt(3)/(2*sqrt(pi)),   -sqrt(3)/(2*sqrt(pi)) }
// Constants for the quadratic polynomials
// g_vCartSHConstB3 = {
//    sqrt(15)/(2*sqrt(pi)),   -sqrt(15)/(2*sqrt(pi)),   sqrt(5)/(4*sqrt(pi)),   sqrt(15)/(4*sqrt(pi)) }
//-----------------------------------------------------------------------------
float4 ComputeZonalCartesianPRTDiffuse(int aLightIdx, float3 oSNorm, float3 vZHCoeff[ NUM_ZH_COEFF] )
{
    float3 Band_12, Band3, Band3_Final;
    float4 sNormB12, sNormB3, cRadiance = 0;

    // Linear + Constant Polynomials
```

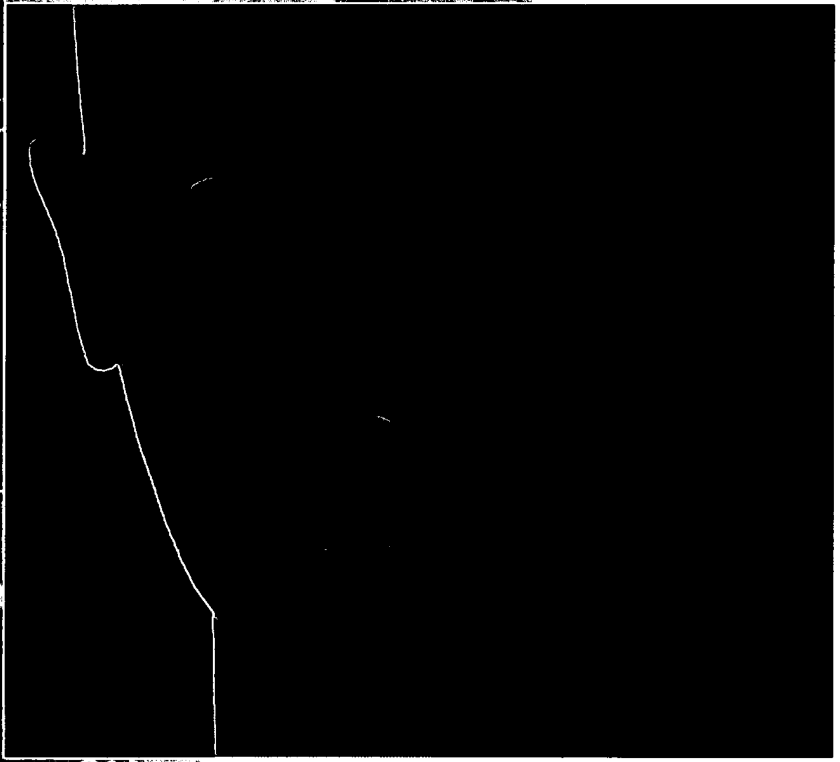
Bands 0 and 1

```
    sNormB12 = float4(1, oSNorm.yzx) * g_vCartSHConstB12;
    Band_12.r = dot(g_vSHLightRed[ aLightIdx][ 0] , sNormB12 * float4(vZHCoeff[ 0] .r, vZHCoeff[ 1] .rrr) );
    Band_12.g = dot(g_vSHLightGreen[ aLightIdx][ 0] , sNormB12 * float4(vZHCoeff[ 0] .g, vZHCoeff[ 1] .ggg) );
    Band_12.b = dot(g_vSHLightBlue[ aLightIdx][ 0] , sNormB12 * float4(vZHCoeff[ 0] .b, vZHCoeff[ 1] .bbb) );
    cRadiance.rgb += Band_12;

    // First 4 Quadratic Polynomials
```

Band 2

```
    sNormB3 = oSNorm.xyzz * oSNorm.yzzx;
    sNormB3.z = (3.0 * sNormB3.z) - 1.0;
    sNormB3 *= g_vCartSHConstB3.xyzy;
    Band3.r = dot(g_vSHLightRed[ aLightIdx][ 1] ,   sNormB3);
    Band3.g = dot(g_vSHLightGreen[ aLightIdx][ 1] , sNormB3);
    Band3.b = dot(g_vSHLightBlue[ aLightIdx][ 1] ,  sNormB3);
    cRadiance.rgb += Band3 * vZHCoeff[ 2] ;

    // Final Quadratic Polynomial
    Band3_Final.rgb = (oSNorm.x * oSNorm.x) - (oSNorm.y * oSNorm.y);
    Band3_Final.rgb *= g_vCartSHConstB3.w;
    Band3_Final.rgb *= vZHCoeff[ 2] ;
    cRadiance.r += g_vSHLightRed[ aLightIdx][ 2] .x * Band3_Final.r;
    cRadiance.g += g_vSHLightGreen[ aLightIdx][ 2] .x * Band3_Final.g;
    cRadiance.b += g_vSHLightBlue[ aLightIdx][ 2] .x * Band3_Final.b;

    return cRadiance * g_vMaterialDiffuseColor;
}
```

- About 23 shader instructions for 3rd order ZH PRT.
  - Can replace per-band SH evaluation of irradiance with a cube map look up for higher order bands.


SH based (CPCA) PRT Lighting

ZH based PRT Lighting

Results Using ZH vs SH

- For the skin material the ZH results are very similar to straightforward SH based PRT.
  - In general using zonal harmonics causes a slight loss in some of the directionally dependant hue shifting due to sub-surface scattering and diffuse interreflections.

References

- [Borshukov03] George Borshukov and J.P. Lewis, “Realistic Human Face Rendering for *The Matrix Reloaded*,” Technical Sketches, SIGGRAPH 2003.
- [Green04] Simon Green, “Real-Time Approximations to Subsurface Scattering,” GPU Gems 2004.
- [Annen04] Tomas Annen, Jan Kautz, Fredo Durand, and Hans-Peter Seidel, “Spherical Harmonic Gradients for Mid-Range Illumination,” Proceedings of Eurographics Symposium on Rendering, June 2004
- [Mertens03] Tom Mertens, Jan Kautz, Philippe Bekaert, Hans-Peter Seidel and Frank Van Reeth, “Efficient Rendering of Local Subsurface Scattering ,” Proceedings of Pacific Graphics 2003 .
- [Sander04] Pedro V. Sander, David Gosselin and Jason L. Mitchell "Real-Time Skin Rendering on Graphics Hardware," SIGGRAPH 2004 Technical Sketch. Los Angeles, August 2004
- [Sloan04] Peter-Pike Sloan and Jason Sandlin, “Practical PRT” Microsoft DirectX Meltdown 2004.

EXHIBIT D

Game Developers Conference

EVOLVE GDC>04

Real Time Skin Rendering

David Gosselin
3D Application Research Group
ATI Research, Inc.

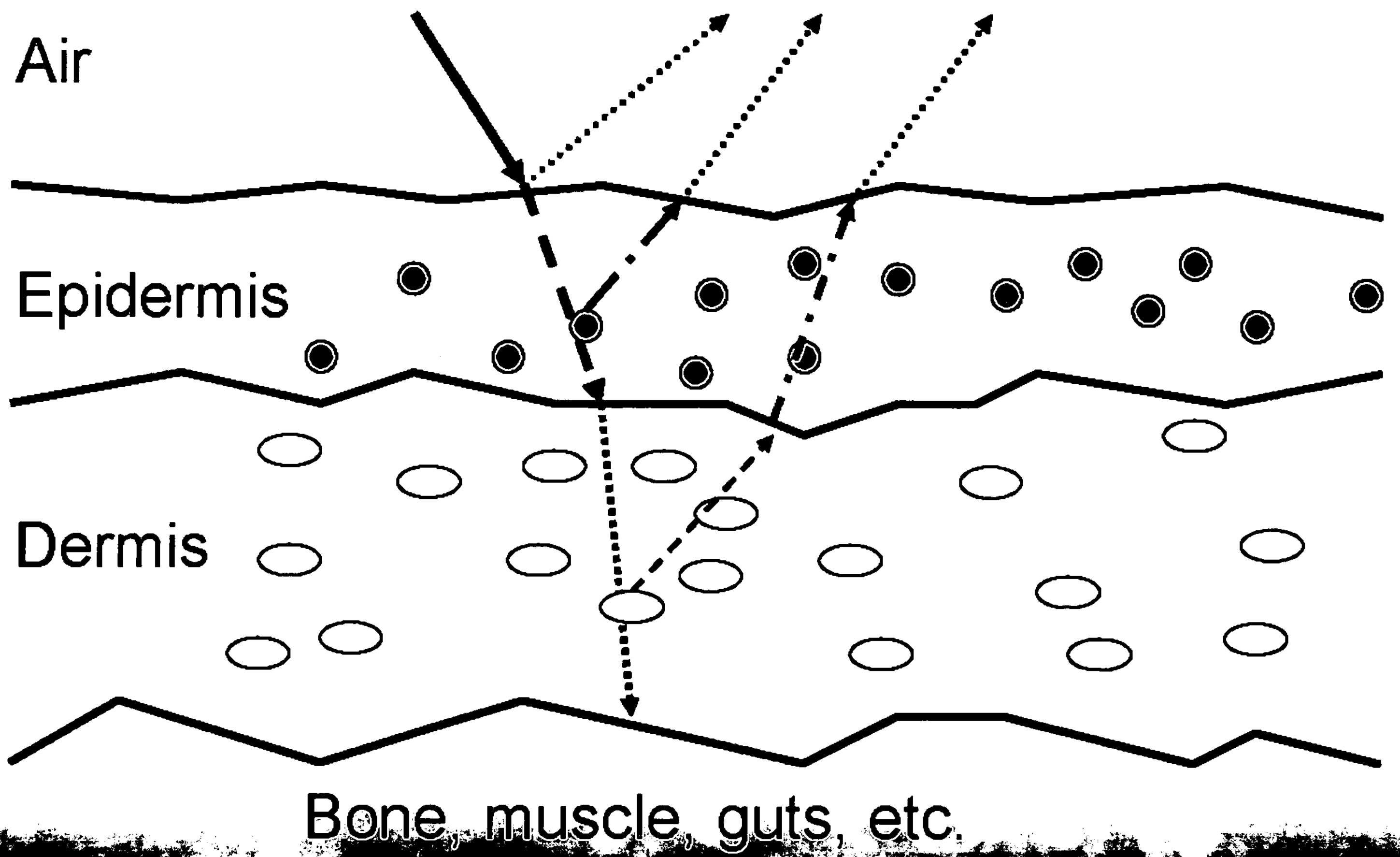
Overview

- Background
- Texture space lighting
- Spatially varying blur
- Dilation
- Adding shadows
- Specular with shadows

Why Skin is Hard

- Most diffuse lighting from skin comes from sub-surface scattering
- Skin color mainly from epidermis
- Pink/red color mainly from blood in dermis
- Lambertian model designed for “hard” surfaces with little sub-surface scattering so it doesn’t work real well for skin

Rough Cross Section



Research

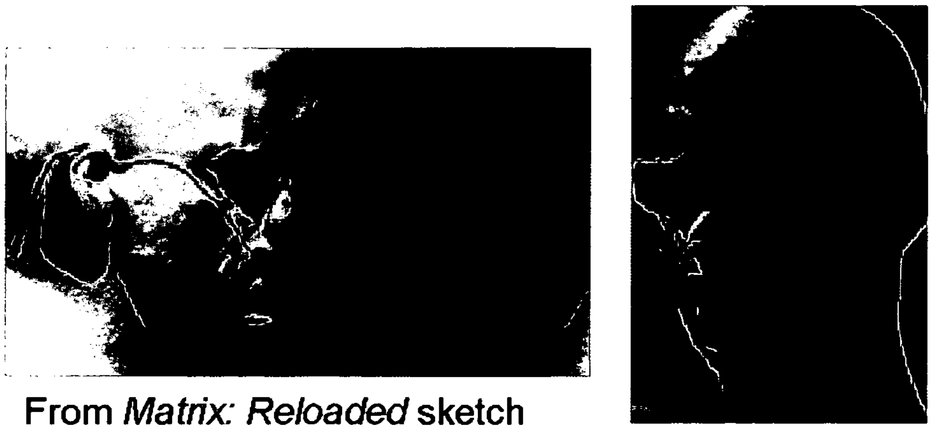
- There are several good mathematical models available
- We looked at using Hanrahan/Krueger (SIGGRAPH 93) based model
  - Good but expensive for current technology
  - Over 100 instructions per light

Basis for Our Approach

- SIGGRAPH 2003 sketch **Realistic Human Face Rendering for "The Matrix Reloaded"** by George Borshukov and J. P. Lewis
- Rendered a 2D light map
- Simulate subsurface diffusion in image domain (different for each color component)
- Used traditional ray tracing for areas where light can pass all the way through (e.g. ears)
- Also capture fine detail normal maps and albedo maps

Texture Space Subsurface Scattering

- From **Realistic Human Face Rendering for "The Matrix Reloaded"** @ SIGGRAPH 2003:


From *Matrix: Reloaded* sketch

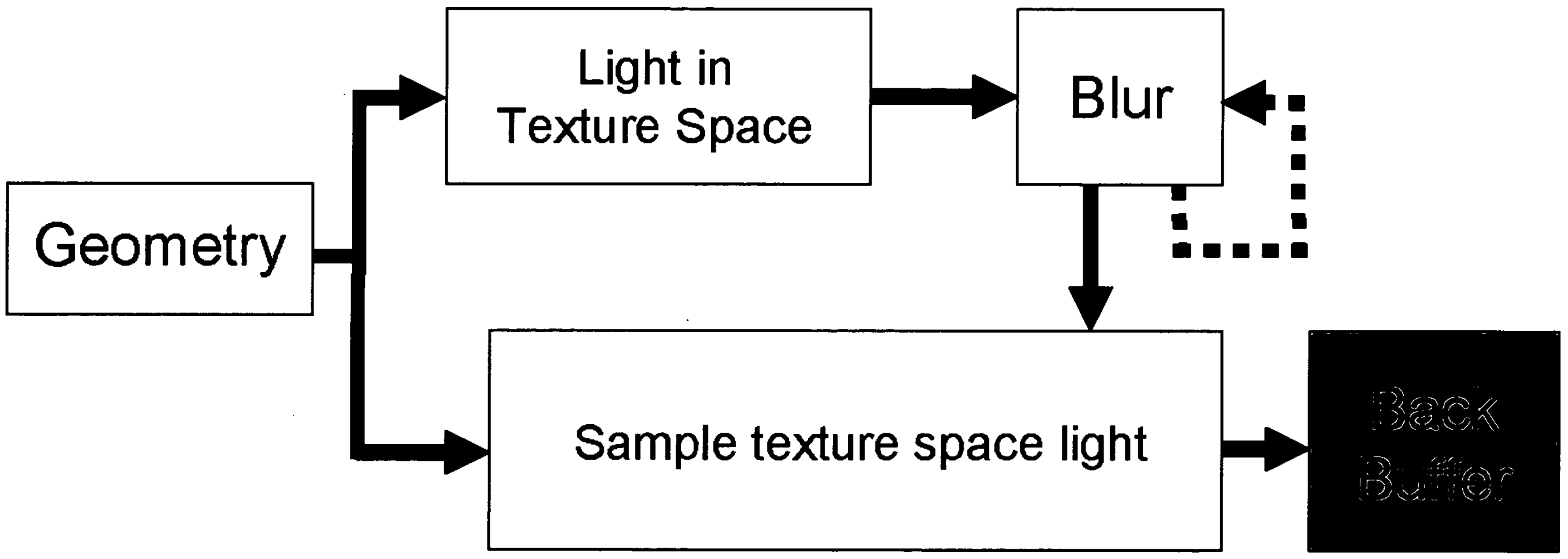
- Our results:

Current skin in Real Time

Texture Space Lighting for Real Time

- Render diffuse lighting into an off-screen texture using texture coordinates as position
- Blur the off-screen diffuse lighting
- Read the texture back and add specular lighting in subsequent pass
- We only used bump map for the specular lighting pass

Basic Approach

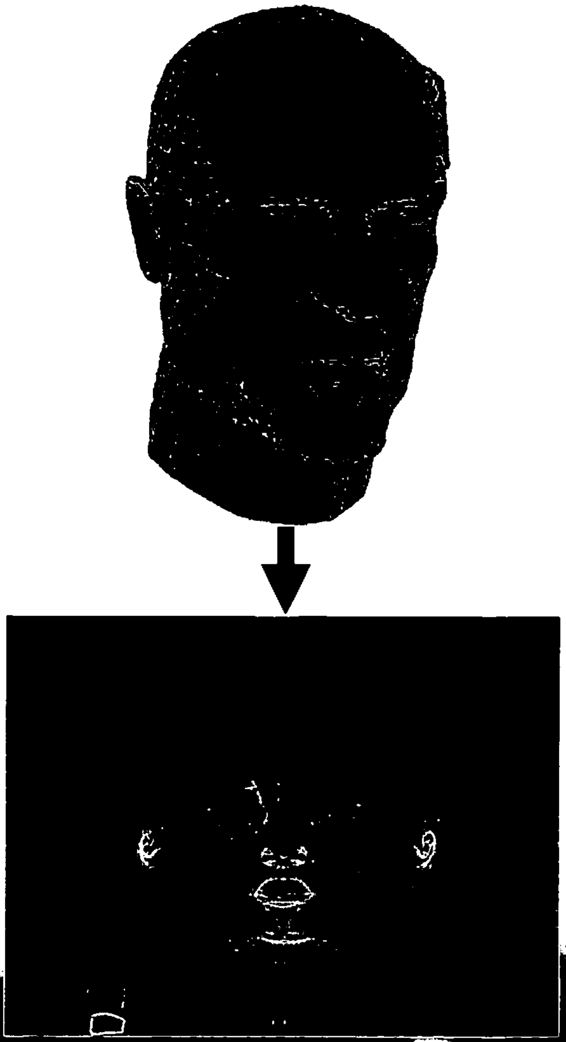


Game Developers Conference

EVOLVE GDC>04

Texture Coordinates as Position

- Need to light as a 3D model but draw into texture
- By passing texture coordinates as "position" the rasterizer does the unwrap
- Compute light vectors based on 3D position and interpolate



Game Developers Conference

EVOLVE GDC>04

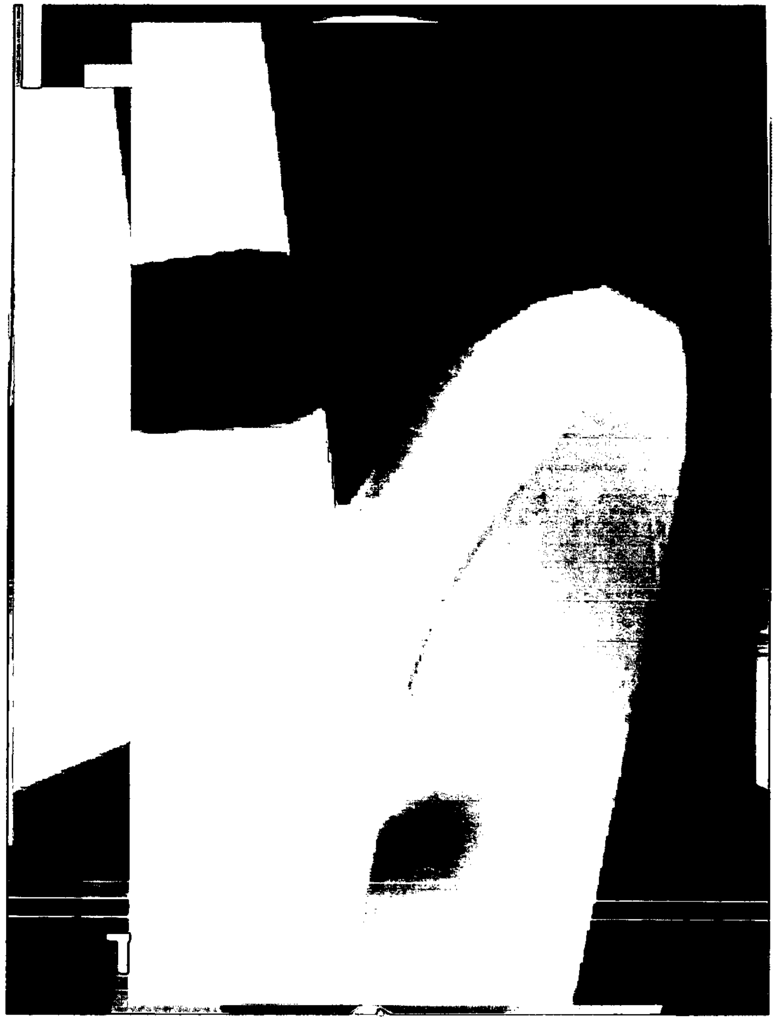
Texture Lighting Vertex Shader

```
VsOutput main (VsInput i)
{
   // Compute output texel position
   VsOutput o;
   o.pos.xy = i.texCoord*2.0-1.0;
   o.pos.z = 1.0;
   o.pos.w = 1.0;

   // Pass along texture coordinates
   o.texCoord = i.texCoord;

   // Skin
   float4x4 mSkinning = SiComputeSkinningMatrix (i.weights, i.indices);
   float4 pos = mul (i.pos, mSkinning);
   pos = pos/pos.w;
   o.normal = mul (i.normal, mSkinning);

   // Compute Object light vectors
   // etc.
```

Texture Lighting Pixel Shader

```
float4 main (PsInput i) : COLOR
{
  // Compute Object Light 0
  float3 vNormal = normalize (i.normal);
  float3 lightColor = 2.0 * SiGetObjectAmbientLightColor(0);
  float3 vLight = normalize (i.oaLightVec0);
  float NdotL = SiDot3Clamp (vNormal, vLight);
  float3 diffuse = saturate (NdotL * lightColor);

  // Compute Object Light 1 & 2
  . . .

  float4 o;
  o.rgb = diffuse;
  float4 cBump = tex2D (tBump, i.texCoord);
  o.a = cBump.a; // Save off blur size
  return o;
}
```

Texture Lighting Results



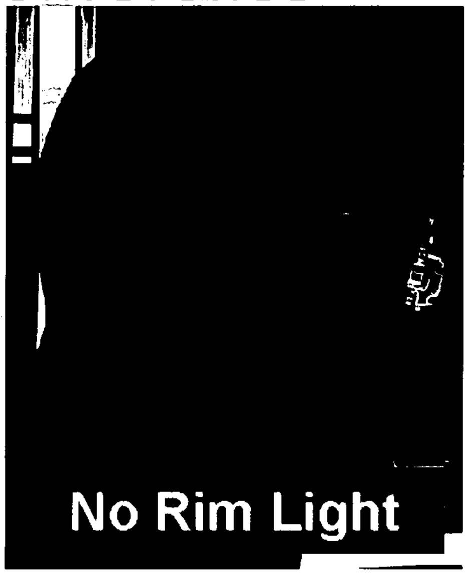
Rim light

- We wanted to further emphasize the light that bleeds through the skin when backlit
- Compute the dot product between the negative light vector and the view vector
- Multiply result by Fresnel term
- Only shows up if there is a light roughly "behind" the object

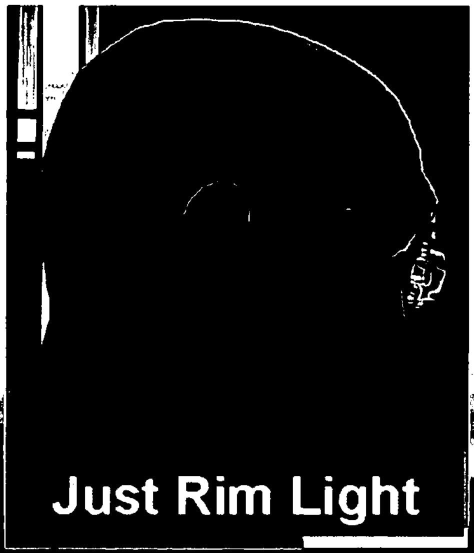
Pixel Shader

```
float4 main (PsInput i) : COLOR
{
   // Normalize interpolated vectors.
   float3 vNormal = normalize (i.normal);
   float3 vView = normalize (i.viewVec);
   float NdotV = SiDot3Clamp (vNormal, vView);
   float fresnel = (1.0f - NdotV);

   // Compute Object Light 0
   float3 lightColor = 2.0 * SiGetObjectAmbientLightColor(0);
   float3 vLight = normalize (i.oaLightVec0);
   float NdotL = SiDot3Clamp (vNormal, vLight);
   float VdotL = SiDot3Clamp (-vLight, vView);
   float3 diffuse = saturate ((fresnel*VdotL+NdotL)*lightColor);

   // Compute Object Light 1 & 2 in the same way
   // Output diffuse and alpha from bump map (blur size)
   . . .
```

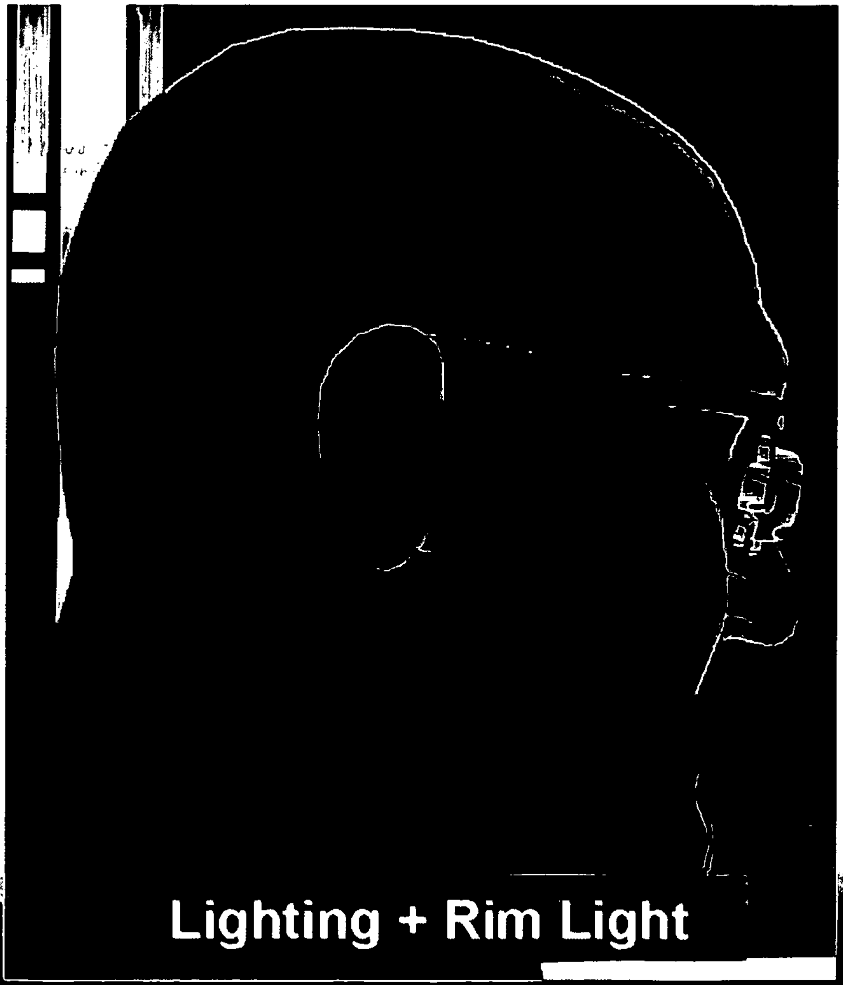
Added Rim Light Result



+



=



Spatially Varying Blur

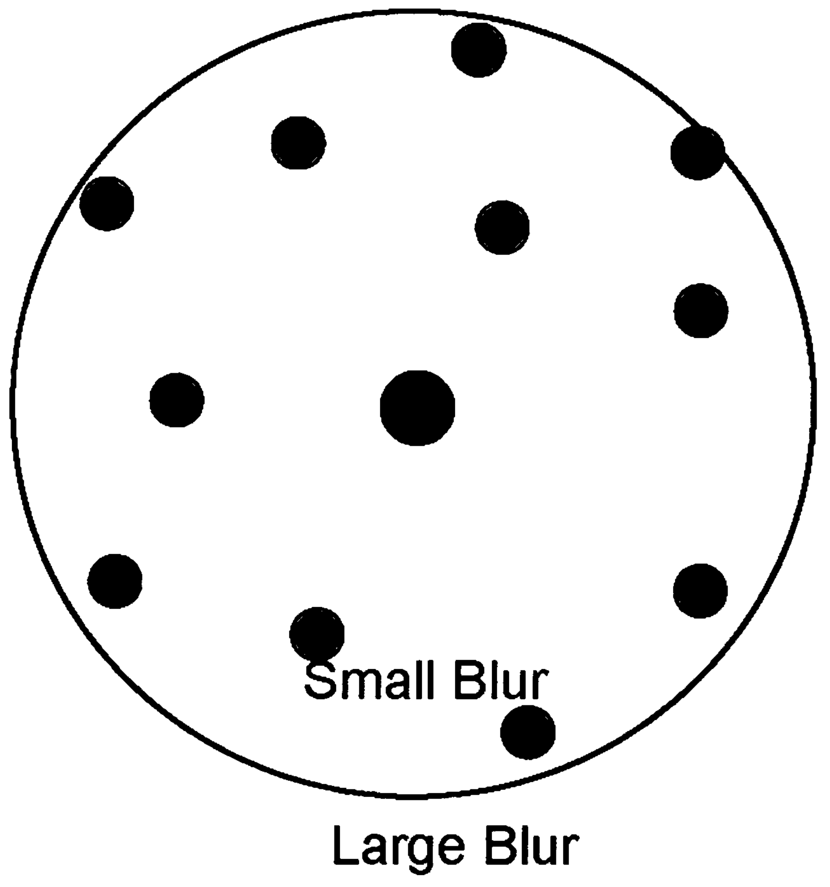
- Used to simulate the subsurface component of skin lighting
- Used a grow-able Poisson disc filter
- Read the kernel size from a texture
- Allows varying the subsurface effect
  – Higher for places like ears/nose
  – Lower for places like cheeks

Growable Filter Kernel

- Stochastic sampling
- Poisson distribution
- Samples stored as 2D offsets from center

● Center Sample

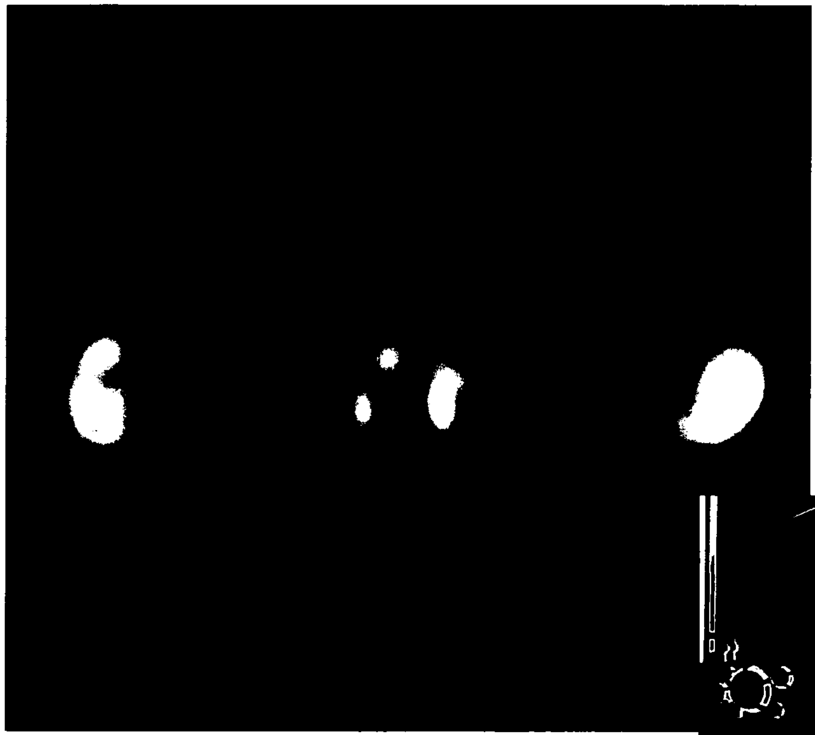
● Outer Samples

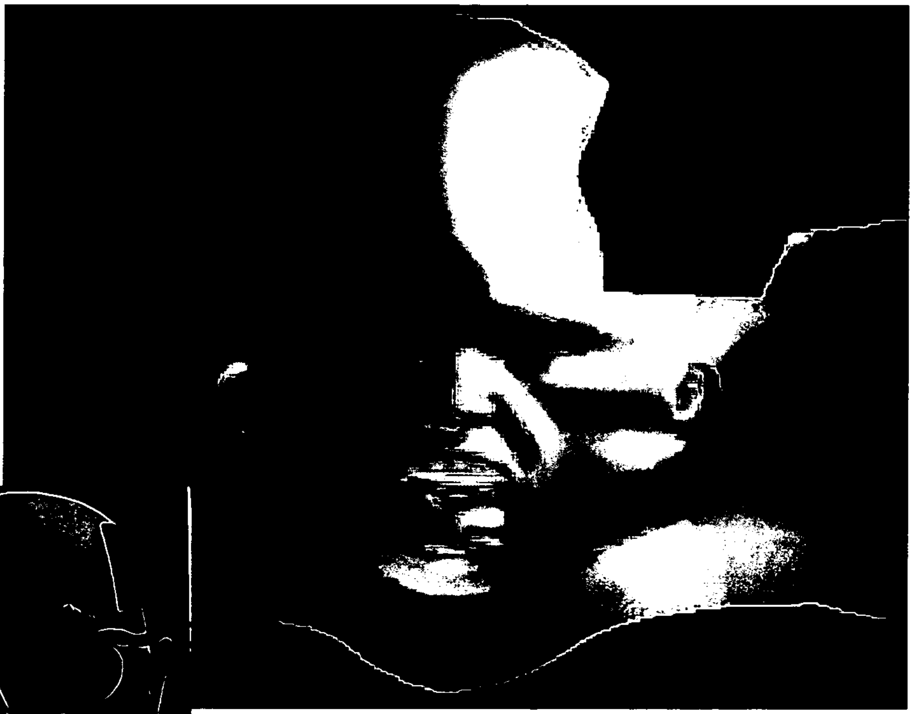


Spatially Varying Blur Pixel Shader

```
float4 main (PsInput i) : COLOR
{
   float2 poisson[12] = . . . // Texel offsets from center

   // Figure out blur size
   float4 center = tex2D(tRenderedScenePong, i.texCoord);
   float blurSize = center.a*vBlurScale.x + vBlurScale.y;

   // Loop over the taps summing contributions
   float3 cOut = center.rgb;
   for (int tap = 0; tap < 12; tap++)
   {
      // Sample using Poisson taps
      float2 coord = i.texCoord.xy+(vPixelSize*poisson[tap]*blurSize);
      float4 sample = tex2D (tRenderedScenePong, coord);
      cOut += sample.rgb;
   }
   return float4(cOut / 13.0f, center.a);
}
```

Blur Size Map and Blurred Lit Texture


Blur Kernel Size Map


Texture Space Lighting

Result

Dilation

- Texture seams can be a problem (unused texels, bilinear blending artifacts)
- During the blur pass we need to dilate
- Use the alpha channel of off-screen texture to determine where we wrote
- If any sample has 1.0 alpha, just copy the sample with the lowest alpha

Dilation + Blur Pixel Shader Code

```
float4 main (PsInput i) : COLOR
{
   float2 poisson[12] = // Texel offsets from center

   // Figure out blur size
   float4 center = tex2D(tRenderedScenePong, i.texCoord);
   float blurSize = center.a*vBlurScale.x + vBlurScale.y;

   // flag is the max alpha value. If it is 1.0f, then sample is
   // close to the boundary since we clear alpha to 1.0
   float flag = center.a;
```

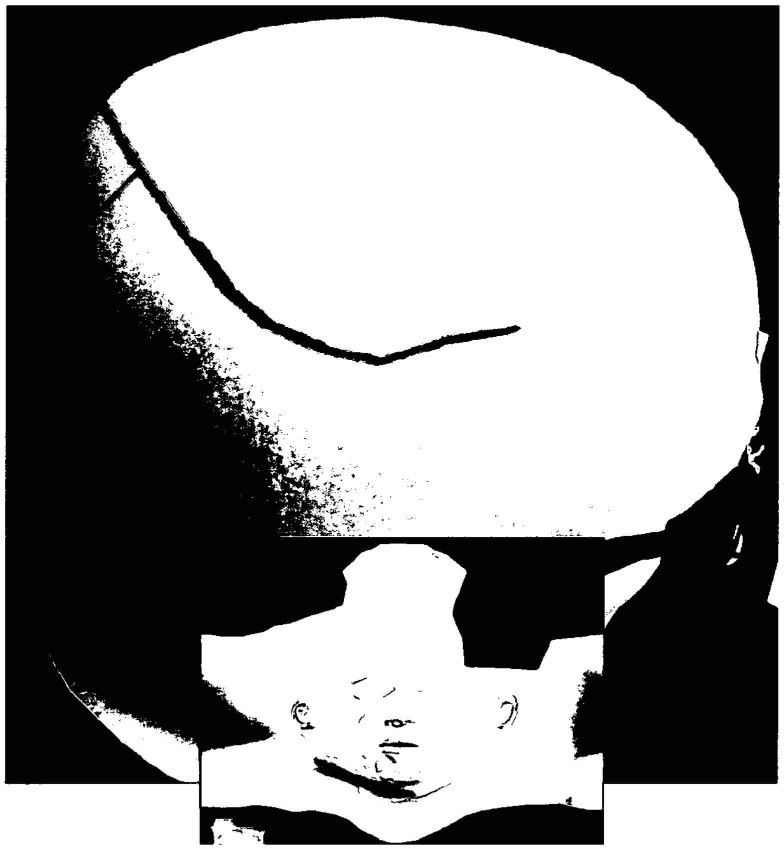
Main Dilate/Blur Pixel Shader Loop

```
// Loop over the taps summing contributions
float3 cOut = center.rgb;
for (int tap = 0; tap < 12; tap++)
{
   // Sample using Poisson distribution
   float2 coord = i.texCoord.xy + (vPixelSize*poisson[tap]*blurSize);
   float4 sample = tex2D (tRenderedScenePing, coord);
   cOut += sample.rgb;

   // Figure out if we need to change the flag
   flag = max (sample.a, flag);
   if (sample.a < center.a)
   {
      // Store texel with lowest alpha; will be used if close to
      // the boundary to "dilate" by picking a more "inside" texel
      center = sample;
   }
}
```

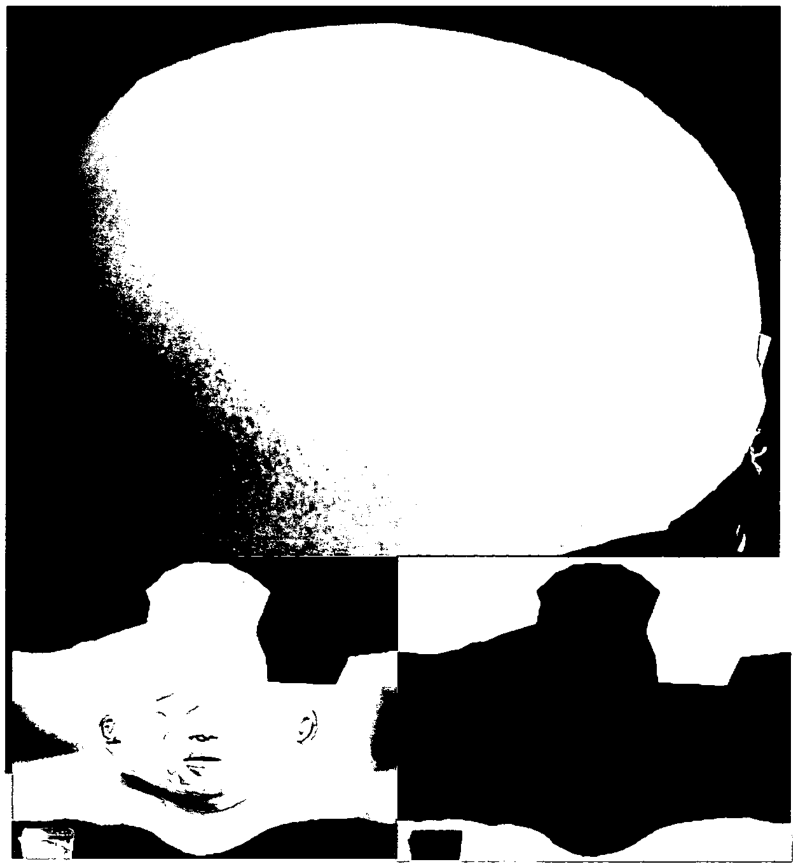
Dilate Test Pixel Shader

```
    // Test the flag to see if we are on a boundary texel
    if (flag == 1.0f)
    {
       // On a boundary pick the texel with the lowest alpha
       return float4 (center.rgb, 1.0f);
    }
    else
    {
       // Not on a boundary same blur as before.
       return float4(cOut / 13.0f, 0.0f);
    }
}
```

Dilation Results



Without Dilation



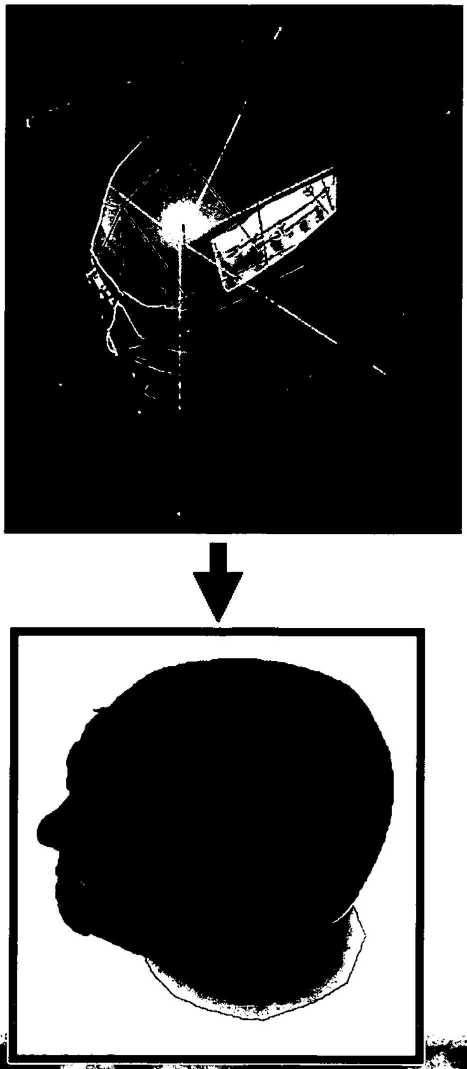
With Dilation

Shadows

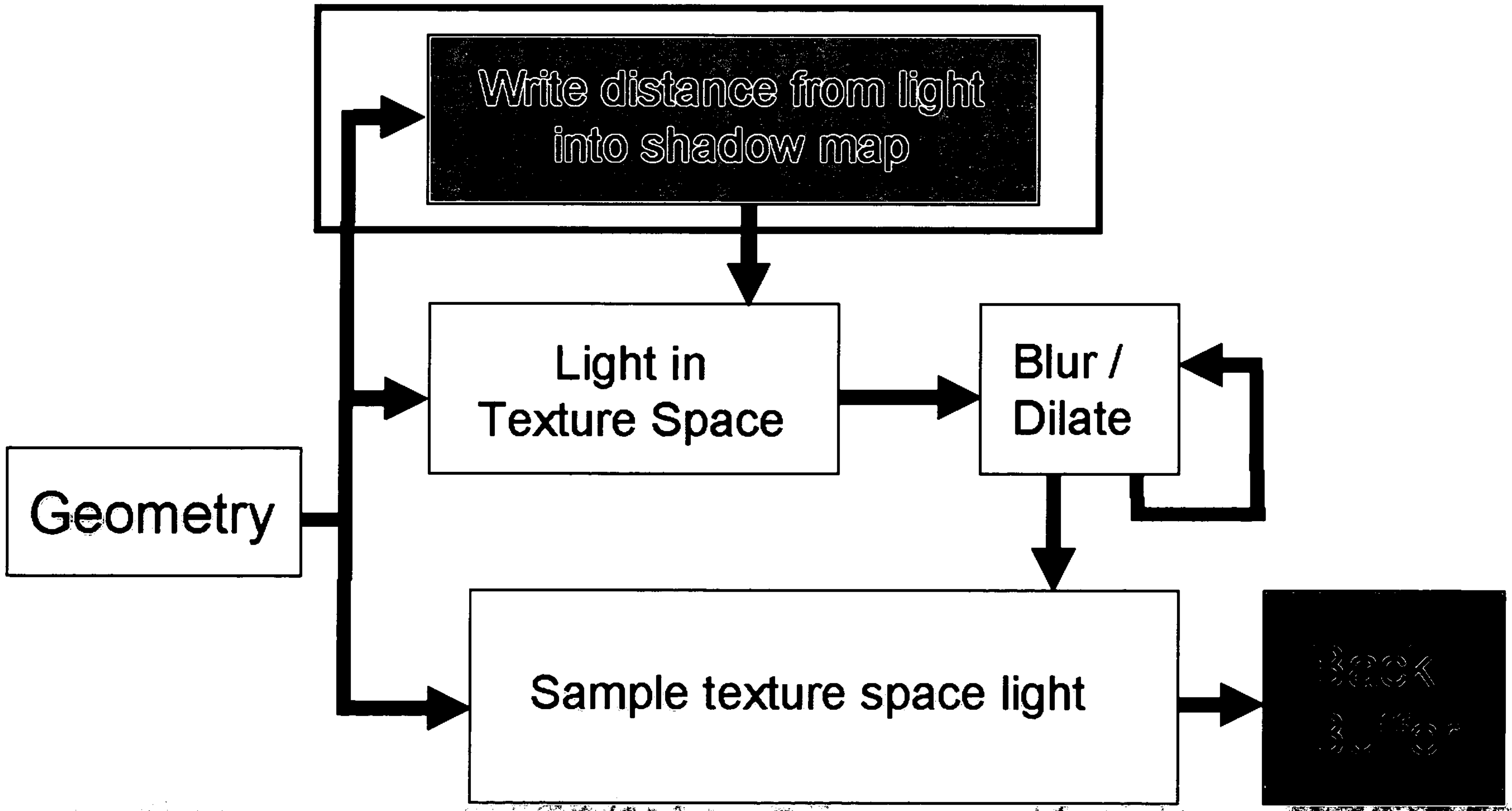
- Used shadow maps
  - Apply shadows during texture lighting
  - Get “free” blur
    - Soft shadows
    - Simulates subsurface interaction
    - Lower precision/size requirements
    - Reduces artifacts
- Only doing shadows from one key light

Shadow Maps

- Create projection matrix to generate map from the light's point of view
- Used bounding sphere of head to ensure texture space is used efficiently
- Write depth from light into off-screen texture
- Test depth values in pixel shader



Texture Lighting With Shadows


Write distance from light into shadow map
Light in Texture Space
Blur / Dilate
Geometry
Sample texture space light
Back Buffer

Shadow Map Vertex Shader

```
float4x4 mSiLightProjection; // Light projection matrix
VsOutput main (VsInput i)
{
   VsOutput o;

   // Compose skinning matrix
   float4x4 mSkinning = SiComputeSkinningMatrix(i.weights, i.indices);

   // Skin position/normal and multiply by light matrix
   float4 pos = mul (i.pos, mSkinning);
   o.pos = mul (pos, mSiLightProjection);

   // Compute depth (Pixel Shader is just pass through)
   float dv = o.pos.z/o.pos.w;
   o.depth = float4(dv, dv, dv, 1);
   return o;
}
```

Texture Lighting Vertex Shader with Shadows

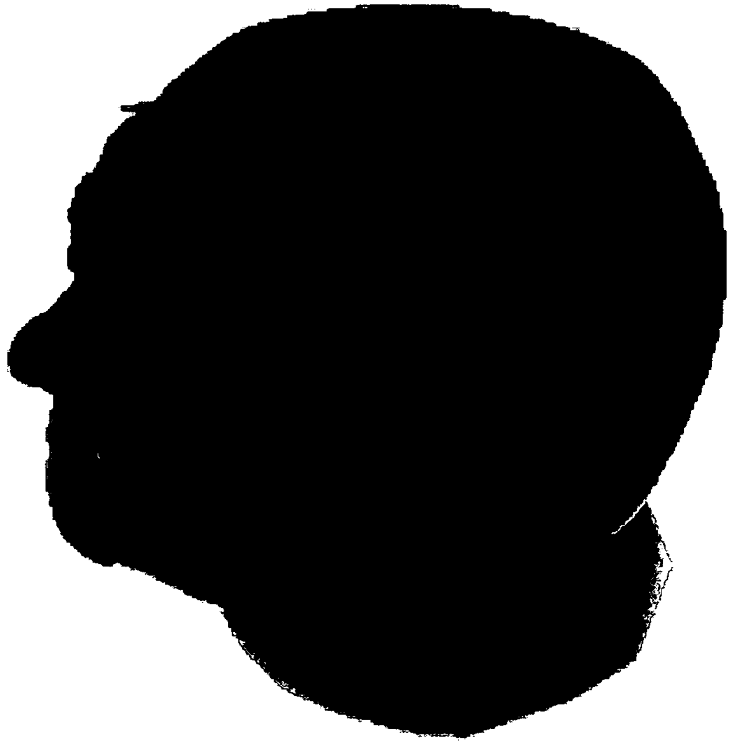
```
VsOutput main (VsInput i)
{
   // Same lead in code as before
   . . .

   // Compute texture coordintates for shadow map
   o.posLight = mul(pos, mSiLightKingPin);
   o.posLight /= o.posLight.w;
   o.posLight.xy = (o.posLight.xy + 1.0f)/2.0f;
   o.posLight.y = 1.0f-o.posLight.y;
   o.posLight.z -= 0.01f;
   return o;
}
```

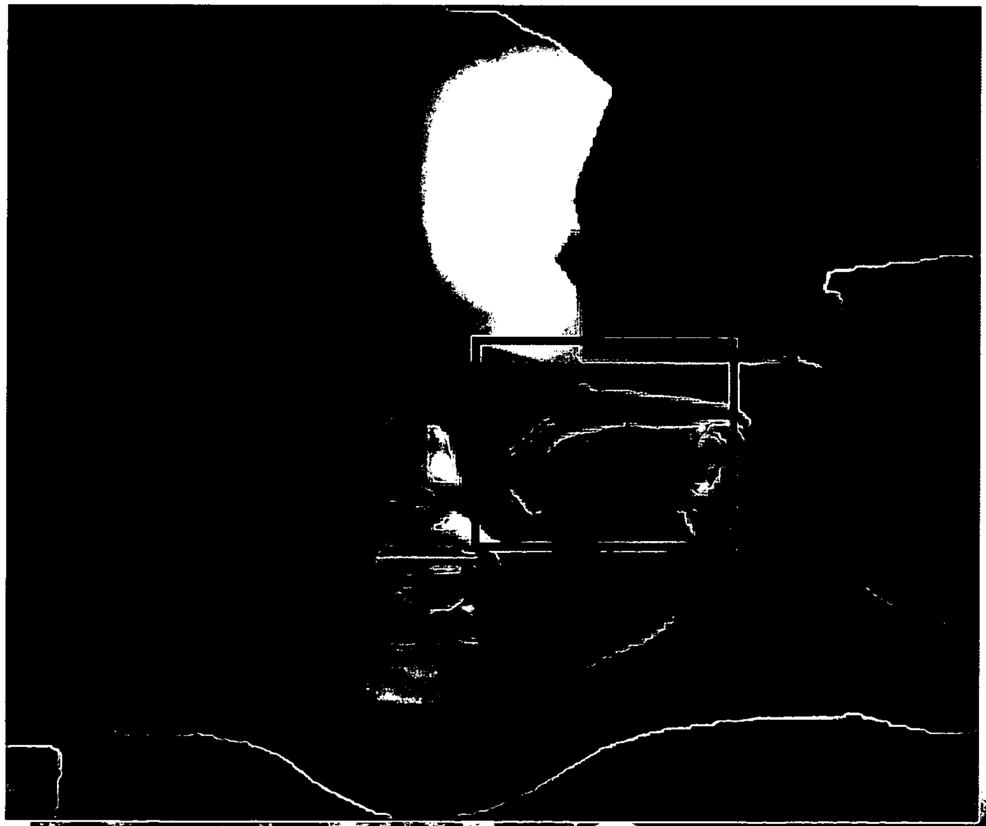
Game Developers Conference

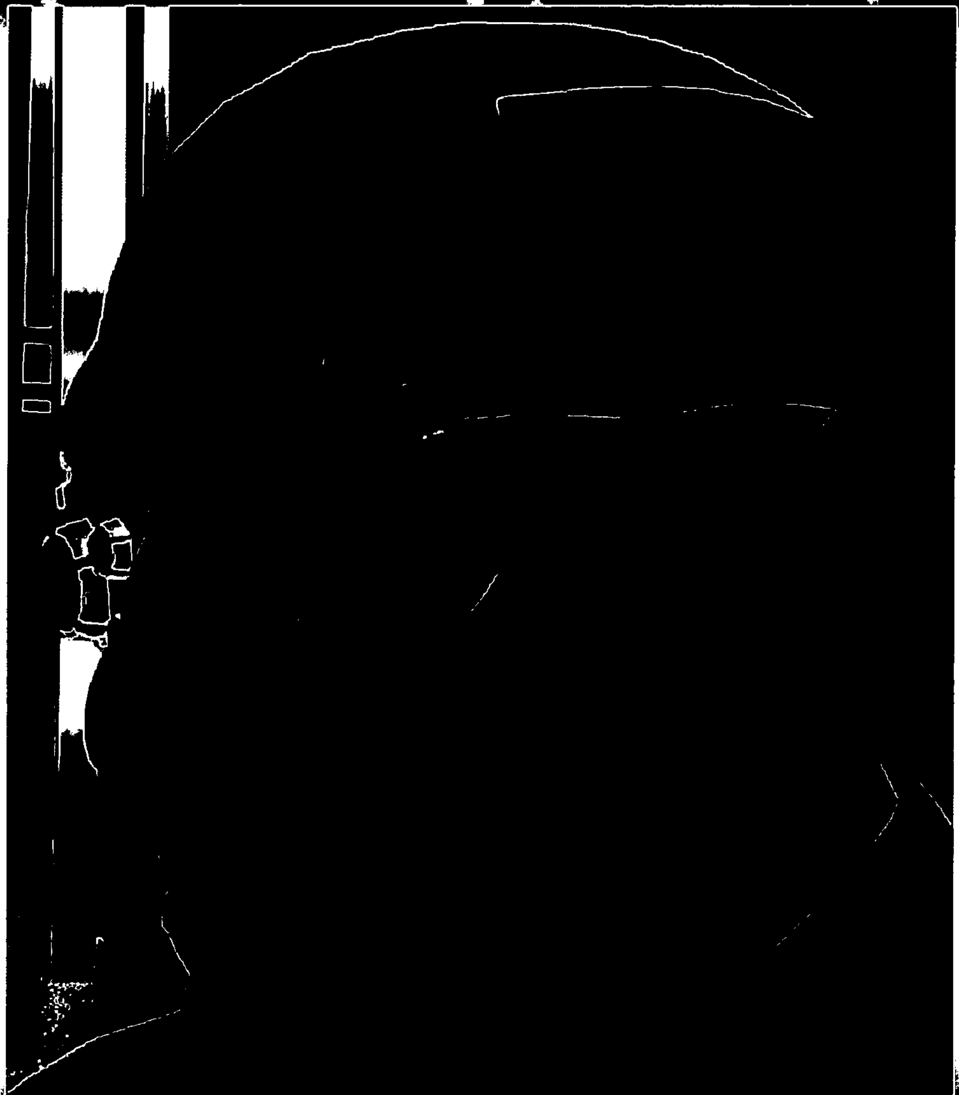
Texture Lighting Pixel Shader with Shadows

```
sampler tShadowMap;
float faceShadowFactor;
float4 main (PsInput i) : COLOR
{
   // Same lead in code
   . . .

   // Compute Object Light 0
   float3 lightColor = 2.0 * SiGetObjectAmbientLightColor(0);
   float3 vLight = normalize (i.oaLightVec0);
   float NdotL = SiDot3Clamp (vNormal, vLight);
   float VdotL = SiDot3Clamp (-vLight, vView);
   float4 t = tex2D(tShadowMap, i.posLight.xy);
   float lfac = faceShadowFactor;
   if (i.posLight.z < t.z) lfac = 1.0f;
   float3 diffuse = lfac * saturate ((fresnel*VdotL+NdotL)*lightColor);

   . . .// The rest of the shader is the same as before
}
```

Shadow Map and Shadowed Lit Texture


Shadow Map (depth)


Shadows in Texture Space

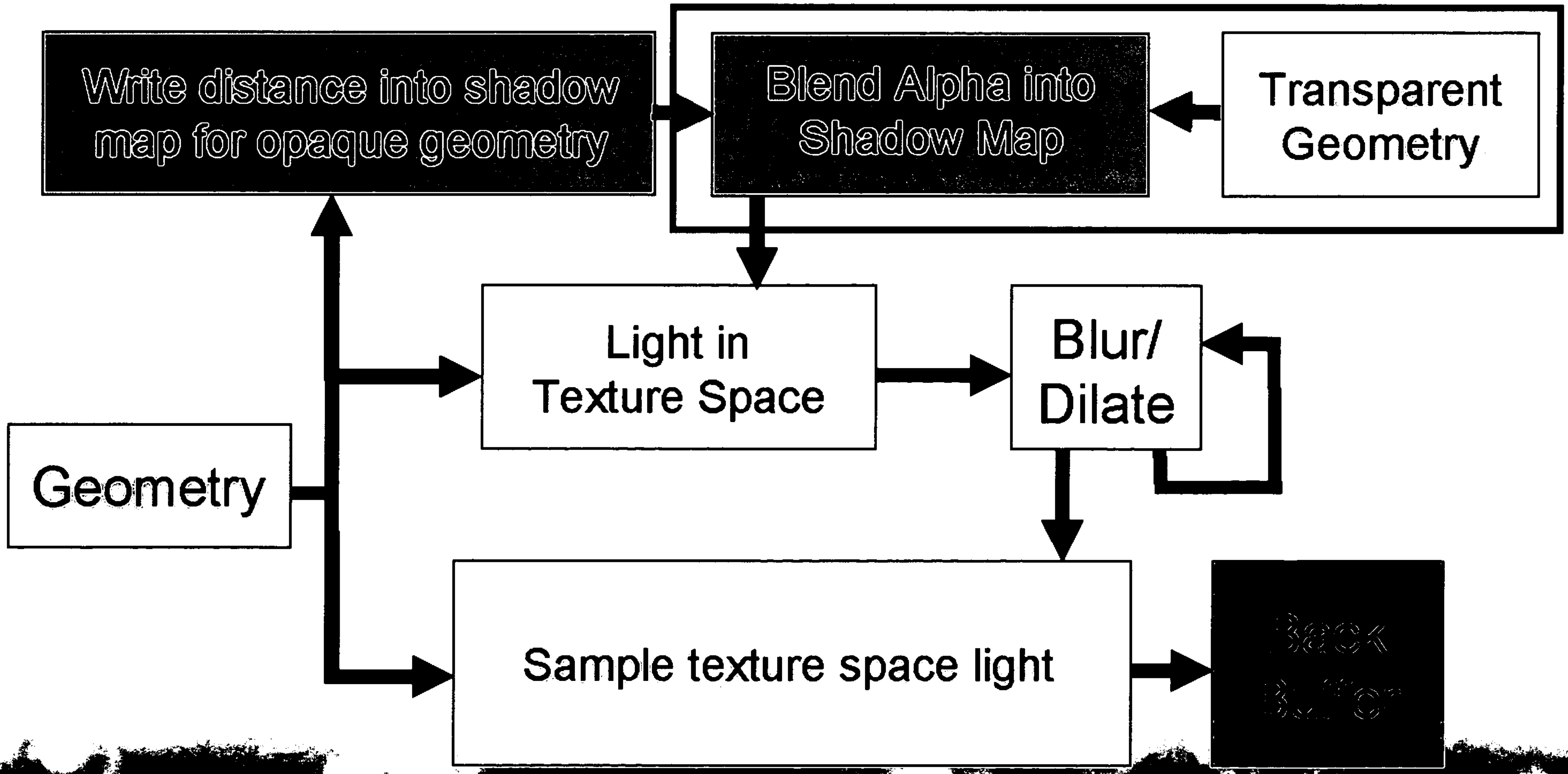


Result with Shadows

Shadows From Translucent Objects

- Allow multiple translucent objects that combine to form opaque shadow (hair)
- Draw opaque shadow geometry first
- Blend alpha of translucent shadow geometry into shadow buffer alpha. Don't write depth!
- In pixel shader: non-shadowed pixels lerp between shadow term and 1.0 based on alpha in shadow map

Texture Lighting With Translucent Shadows


Write distance into shadow map for opaque geometry
Blend Alpha into Shadow Map
Transparent Geometry
Geometry
Light in Texture Space
Blur/ Dilate
Sample texture space light
Back Buffer

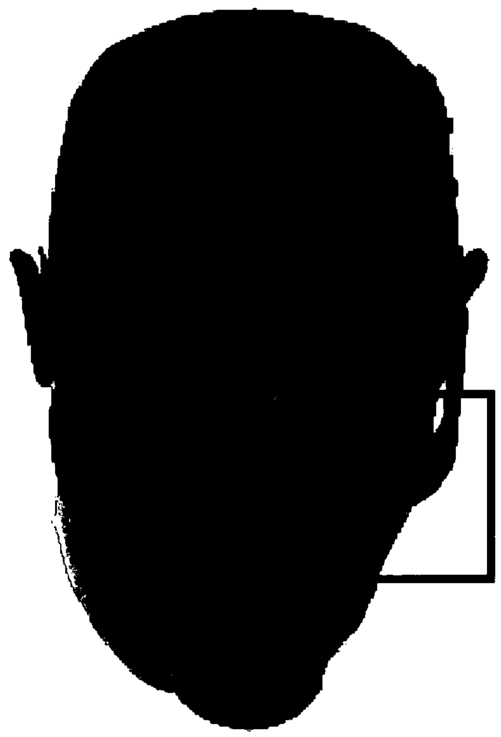
Translucent Shadow Pixel Shader

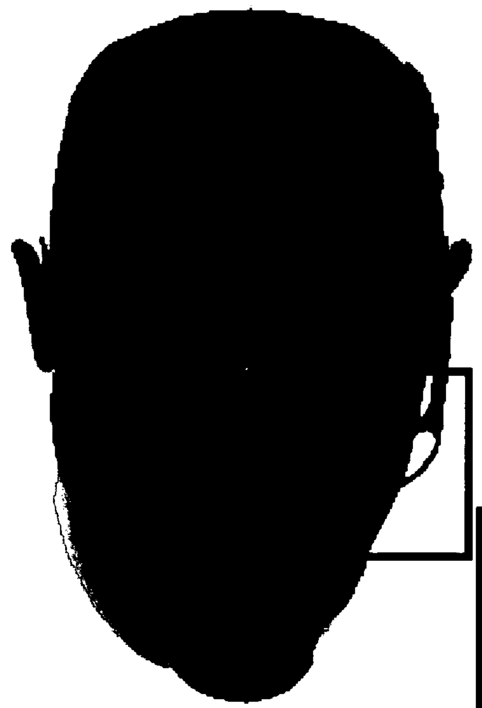
```
float shadowAlpha;
float4 main (PsInput i) : COLOR
{
   // Same lead in
   . . .

   // Usual light 0 code
   . . .
   float4 t = tex2D(tShadowMap, i.posLight.xy);
   float lfac = faceShadowFactor;
   if (i.posLight.z < t.z)
   {
      float alpha = pow(t.a, shadowAlpha);
      lfac = lerp(faceShadowFactor, 1.0f, alpha);
   }
   float3 diffuse = lfac * saturate((fresnel*VdotL+NdotL)*lightColor);

   . . . // Rest of the shader is the same as well
```

Game Developers Conference

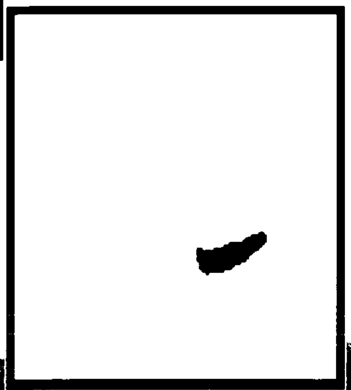
EVOLVE GDC>04

Shadow Map for Transparent Shadows

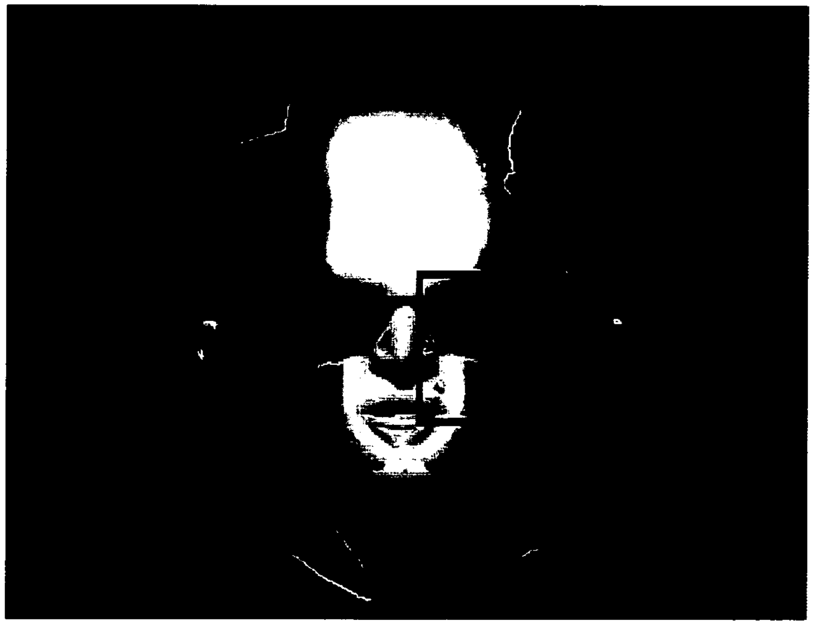

Shadow Map Fully Opaque

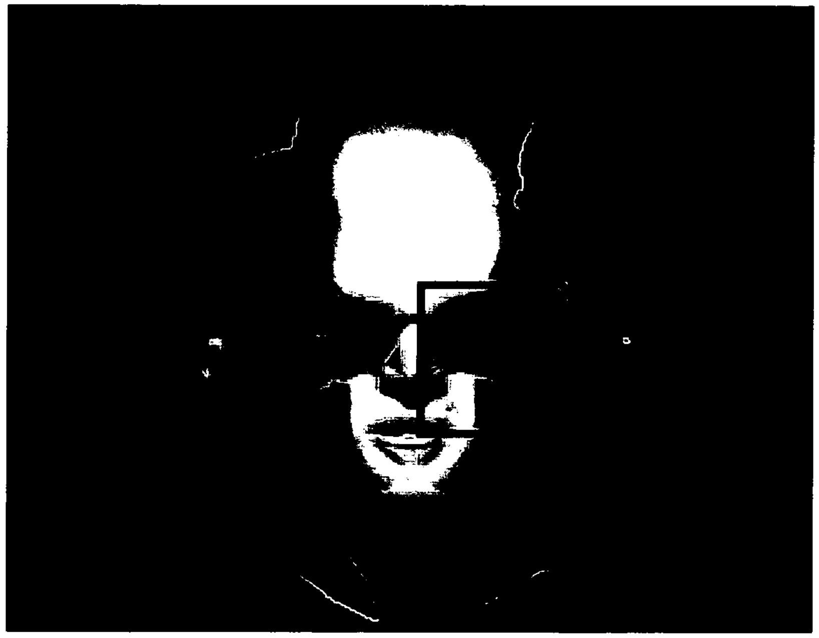

Shadow Map With Translucency

Alpha



Off Screen Light Textures with Translucent Shadows


Opaque Shadows


Translucent Shadows

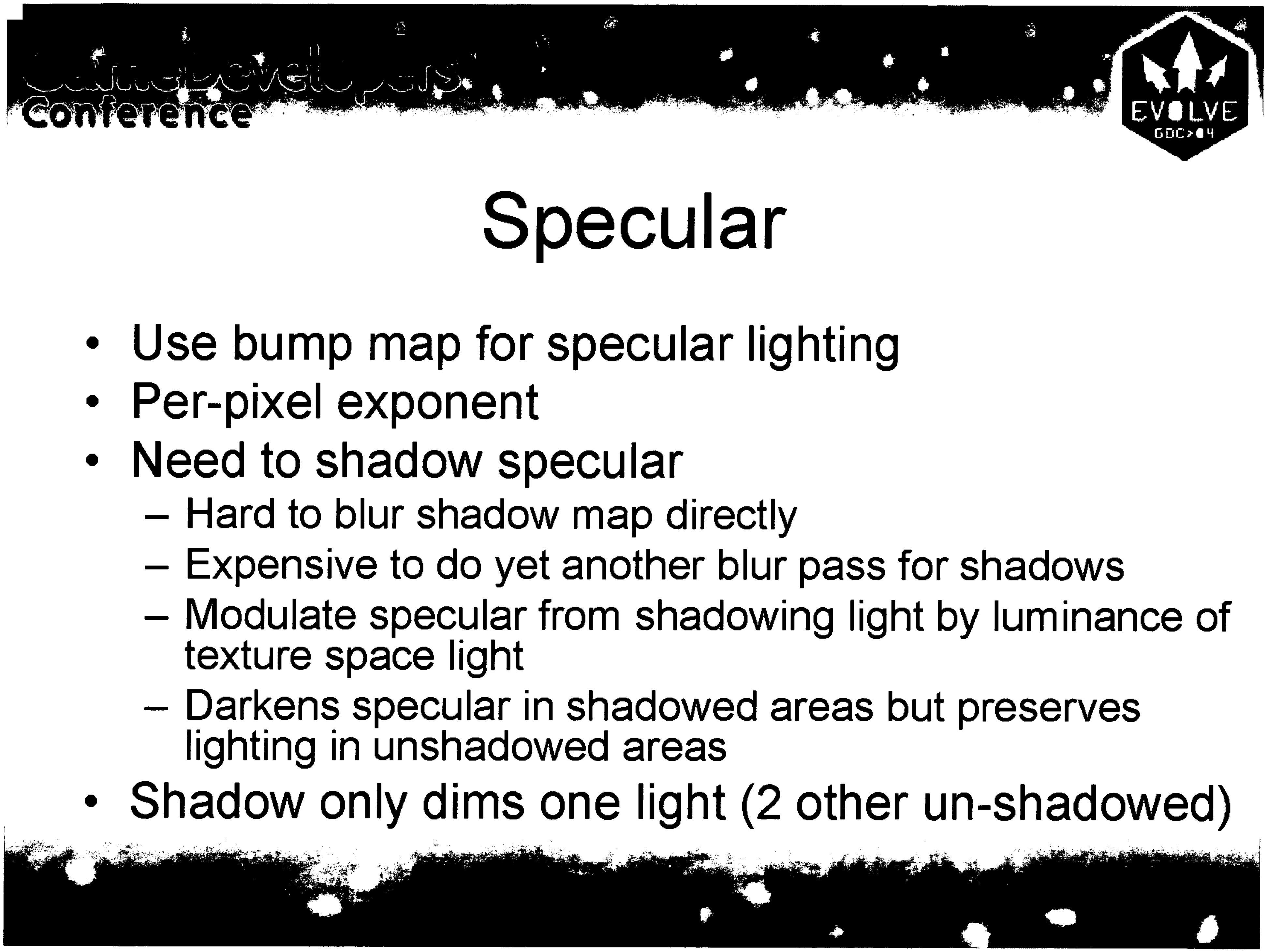
Game Developers Conference

EVOLVE GDC>04

Translucent Shadows Results

Opaque Shadows

Translucent Shadows



Specular

- Use bump map for specular lighting
- Per-pixel exponent
- Need to shadow specular
  - Hard to blur shadow map directly
  - Expensive to do yet another blur pass for shadows
  - Modulate specular from shadowing light by luminance of texture space light
  - Darkens specular in shadowed areas but preserves lighting in unshadowed areas
- Shadow only dims one light (2 other un-shadowed)

Final Pixel Shader (with specular)

```
sampler tBase;
sampler tBump;
sampler tTextureLit;
float4 vBumpScale;
float specularDim;
float4 main (PsInput i) : COLOR
{
   // Get base and bump map
   float4 cBase = tex2D (tBase, i.texCoord.xy);
   float3 cBump = tex2D (tBump, i.texCoord.xy);

   // Get bumped normal
   float3 vNormal = SiConvertColorToVector (cBump);
   vNormal.z = vNormal.z * vBumpScale.x;
   vNormal = normalize (vNormal);
```

Final Pixel Shader

```
// View, reflection, and specular exponent
float3 vView = normalize (i.viewVec);
float3 vReflect = SiReflect (vView, vNormal);
float exponent = cBase.a*vBumpScale.z + vBumpScale.w;
```

```
// Get "subsurface" light from lit texture.
float2 iTx = i.texCoord.xy;
iTx.y = 1-i.texCoord.y;
float4 cLight = tex2D (tTextureLit, iTx);
float3 diffuse = cLight*cBase;
```

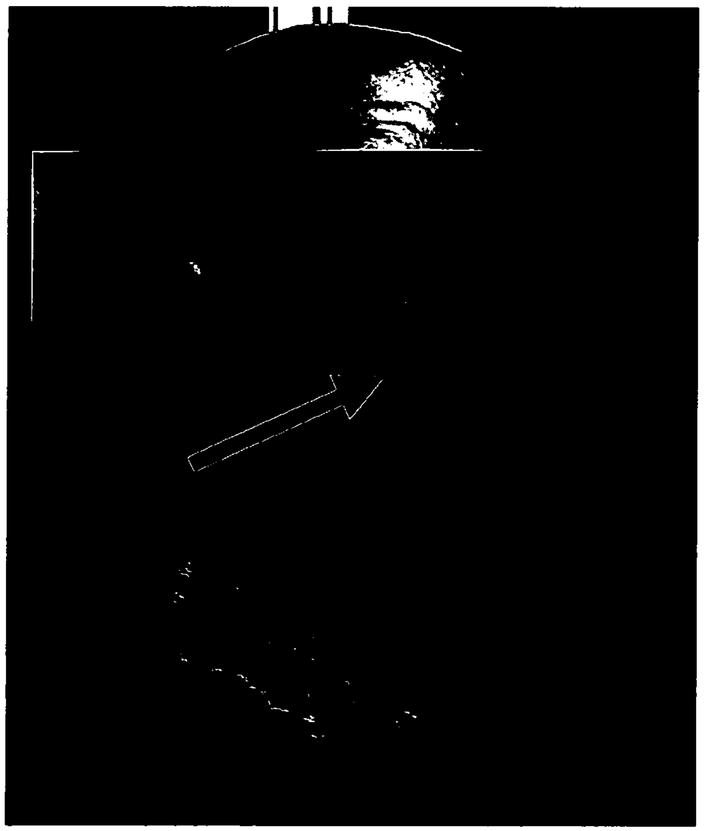
Final Pixel Shader

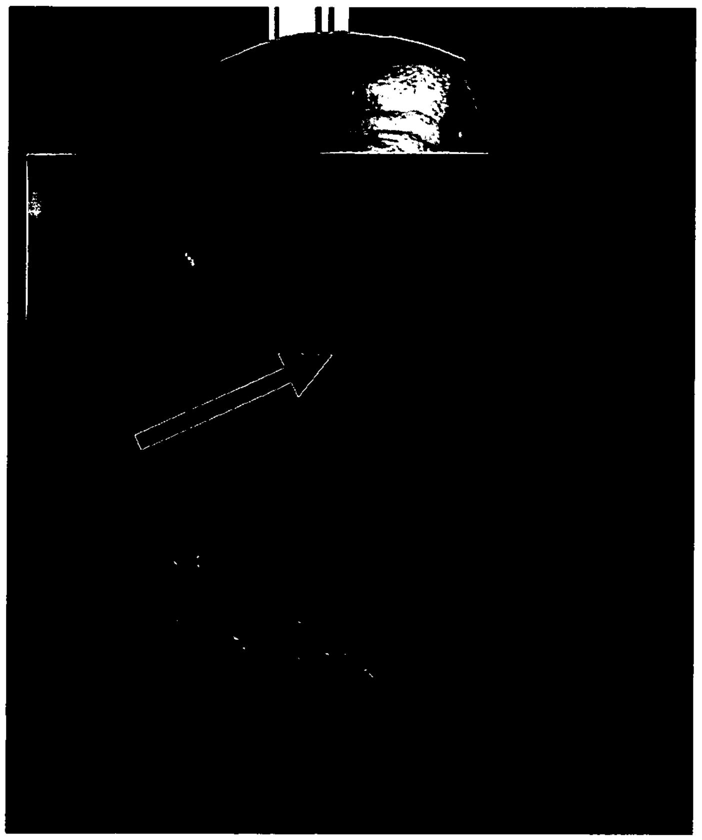
```
    // Compute Object Light 0
    float3 lightColor = 2.0 * SiGetObjectAmbientLightColor(0);
    float3 vLight = normalize (i.oaLightVec0);
    float RdotL = SiDot3Clamp (vReflect, vLight);
    float shadow = SiGetLuminance (cLight.rgb);
    Shadow = pow(shadow, 2);
    float3 specular = saturate(pow(RdotL,exponent)*lightColor)*shadow;

    // Compute Object Light 1 & 2 (same as above but no shadow term)
    . . .

    // Final color
    float4 o;
    o.rgb = diffuse + specular*specularDim;
    o.a = 1.0;
    return o;
}
```

GameDevelopers Conference

EVOLVE GDC>04

Specular Shadow Dim Results

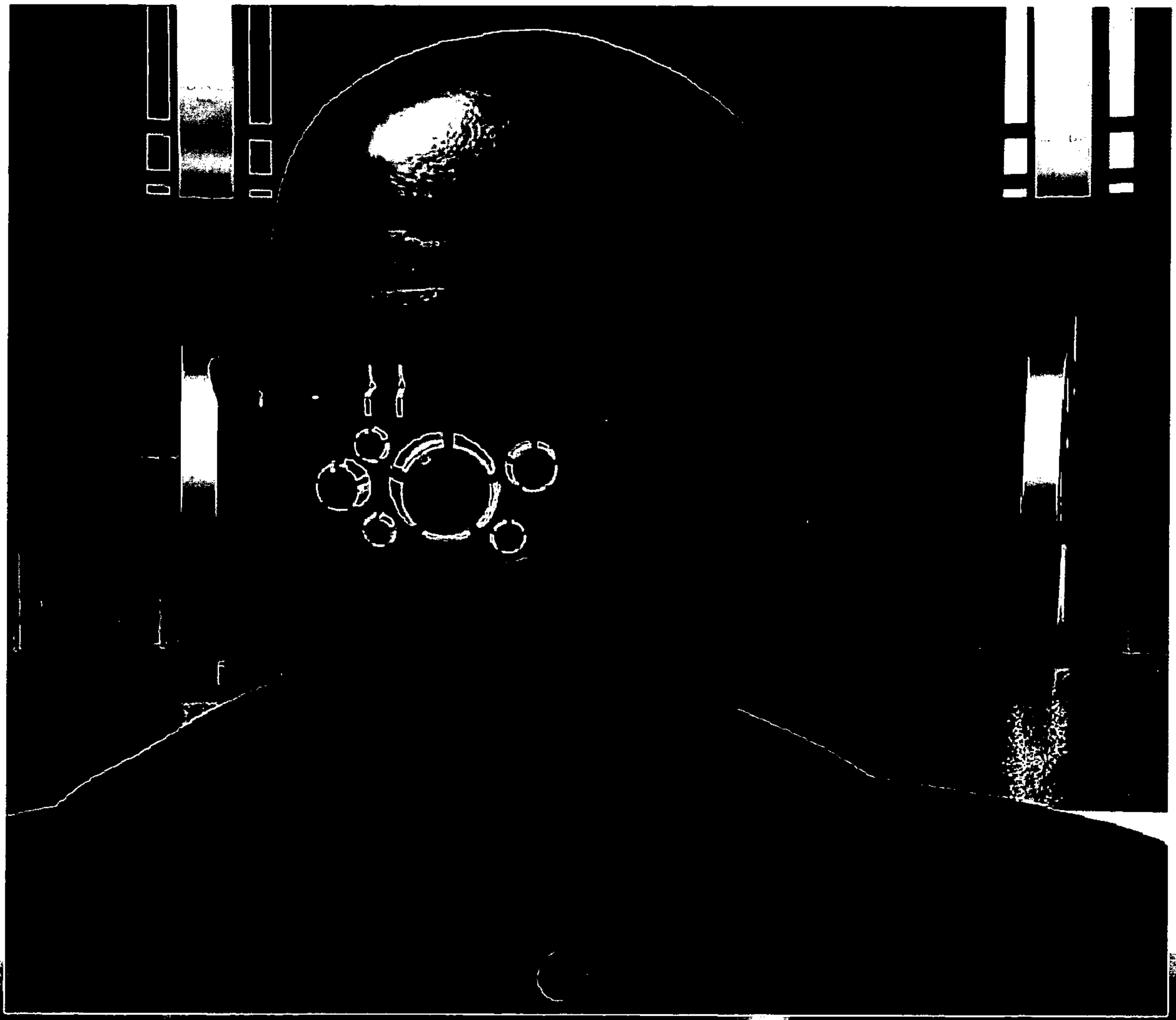

Specular Without Shadows


Specular With Shadows

Game Developers Conference

EVOLVE GDC>04

Demo



Questions?